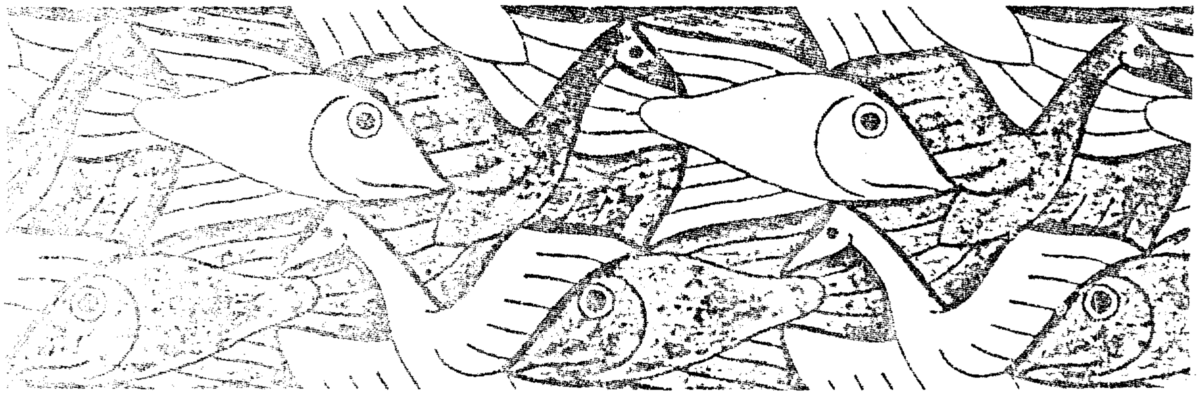
VISION ALGORITHMS AND PSYCHOPHYSICS (FINAL REPORT) 1983 – 1989

Investigator:
(617) 253-5776
WHIT@EGO.MIT.EDU
Contract No. F49620-83-C-0135

Whitman Richards E10-120
Massachusetts Institute of Technology
Cambridge, MA 02139



Abstract: Representing shapes in a manner suitable for recognition has been a challenge for machine vision. Here we approach this problem by combining studies of representations used by the human visual system with computational studies of how such representations can be derived and manipulated by machine. Both axial-based and contour-based descriptors were investigated, with emphasis on the role of curvature which was found to be an important primitive underlying both types of representations. Related, but unreported, studies include color and motion, which often serve as the "glue" that allows one to form appropriate groupings of broken image contours or tokens ~~(see bibliography)~~. This research has yielded over fifty publications, with only the major thrust summarized here.

Key words: Image understanding, shape recognition, visual pattern recognition, visual psychophysics, vision algorithms.

October 1989

[illegible]

Contents

(Front piece panel from Escher.)

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION: Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 89-1883
6a NAME OF PERFORMING ORGANIZATION: Mass. Institute of Technology; 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION: Air Force Office of Scientific Research
6c ADDRESS (City, State, and ZIP Code): Dept. of Brain & Cognitive Sciences, Cambridge, MA 02139	7b ADDRESS (City, State, and ZIP Code): Bolling Air Force Base, Washington, DC 20332
8a NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b OFFICE SYMBOL (If applicable): NL	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: AFOSR-86-0139
8c ADDRESS (City, State, and ZIP Code): Bolling Air Force Base, Washington, DC 20332	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO 61102F; PROJECT NO 2313; TASK NO A5; WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
Vision Algorithms and Psychophysics

12 PERSONAL AUTHOR(S)
Whitman Richards

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Final	FROM 1 Apr 88 TO 31 Aug 89	October 1989	78

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Image understanding, shape recognition, visual pattern recognition, visual psychophysics, vision algorithms.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Representing shapes in a manner suitable for recognition has been a challenge for machine vision. Here we approach this problem by combining studies of representations used by the human visual system with computational studies of how such representations can be derived and manipulated by machine. Both axial-based and contour-based descriptors were investigated, with emphasis on the role of curvature which was found to be an important primitive underlying both types of representations. Related, but unreported, studies include color and motion, which often serve as the "glue" that allows one to form appropriate groupings of broken image contours or tokens (see bibliography). This research has yielded over fifty publications, with only the major thrust summarized here.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT: [✓ UNCLASSIFIED/UNLIMITED] [✓ SAME AS RPT] ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL: J. Tangney	22b TELEPHONE (Include Area Code): (202) 767-5021; 22c OFFICE SYMBOL: NL

DD Form 1473, JUN 86 Previous editions are obsolete SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

90 01 04 145

Vision Algorithms and Psychophysics

1.0 Introduction

"Seeing" requires the construction of symbolic descriptions of the external world. The most useful symbolic descriptions will be representations for each of the various objects in the three-dimensional scene. These objects, in turn, may be broken down further into more detailed modular representations that may include the various attributes of each object such as its color, texture, or the shape and relative motion of its parts. These latter properties are thus our basic building blocks from which more complicated descriptions are built. Vision understanding requires showing how such object properties can be represented internally, and how they can be brought together to create a description suitable for recognition or manipulation. This then, is our goal: to propose and implement a scheme for representing 3D shapes in a manner suitable for recognition.

To reach this goal, the research has proceeded along several parallel, but complementary tracks. The first is the development of a theory for representing 3D shapes, or their 2D projections onto the image. Here we offer two possibilities for representing 3D shapes (Gaussian curvature and Process-based descriptions) and also two proposals for representing 2D shapes (curvature-based codons and the alignment method). None of these theories necessarily excludes using the other. One element common to all the proposals is the key role played by curvature or curvature extrema—a role supported by our psychophysical studies.

All present theories of shape recognition require that the shape be isolated from its background. Hence a second research track has been the identification of candidate "objects" in images, using color, visual motion, stereo, or significant groupings of special image features (such as co-circular or parallel segments). Again, as in the shape recognition proposals, the plausibility of our tentative solutions are supported both by machine implementations and by visual psychophysics.

2.0 Representing 2D Shapes

Line drawings and silhouettes testify to the power of the shape of a contour as a means for object recognition. As illustrated by Attneave's famous "cat" (and also Figure 1), much of this information about shape is carried by the curvature extrema (Attneave, 1954; Fischler & Bolles, 1983; Resnikoff, 1987). One reason why curvature extrema are critical to shape recognition is that they divide the outline into its parts (Hoffman & Richards, 1982; 1984). As illustrated in Figure 2, when objects are created by interpenetrating parts or protrusions, a curvature extrema in the form of a concavity will appear in the image. Such

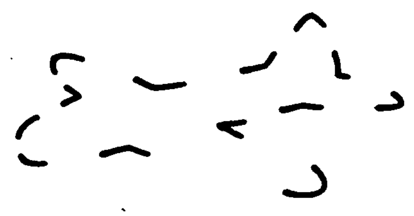


Figure 1 Curvature extrema carry significant information about shape.

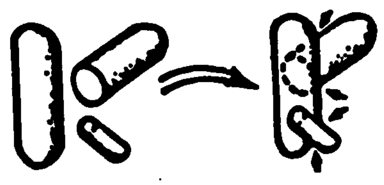


Figure 2 The definition of a part boundary is derived from the transversal property that two interpenetrating surfaces will meet at a concave discontinuity.

concavities thus provide a powerful index into any part-based object recognition scheme.

A second reason for emphasizing curvature as a shape primitive is that the sign of curvature of a silhouette's contour in the image immediately tells the viewer the intrinisic 3D shape of the object in the world (Koenderink & van Doorn, 1980, 1981, 1982, 1987). For example, a piece of silhouette having positive curvature must arise from a 3D surface with positive Gaussian curvature. Such imaging properties mean that very powerful inferences about 3D shapes can be made directly from 2D image contours. Consequently, in 1982 Hoffman and Richards proposed that the first abstract description of the 2D image curves

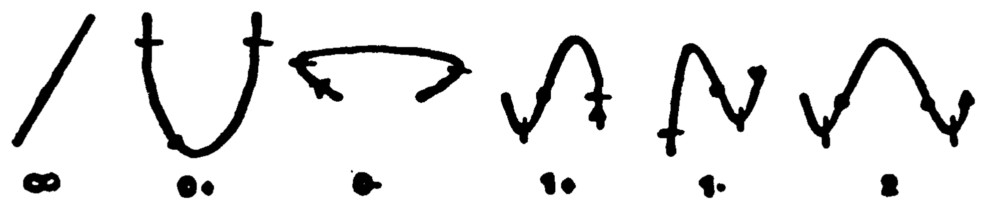


Figure 3 The primitive codon types. Zeros of curvature are indicated by dots, minima by slashes. The straight line (∞) is a degenerative case included for completeness, although it is not treated in the text. (See Richards & Hoffman, 1984, for definitions.)

should be based upon the singularities of curvature—the positive and negative extrema and the inflections of zero curvature. Noting that the extrema of negative curvature (i.e. the concavities) correspond to the part boundaries, it was easy to show that there would only be five primitive or different elemental shapes for representing smooth plane curves. These were called "codons" (Figure 3). Any smooth curve could then be characterized quite simply in terms of the sequence of its codons. Curves with identical codon sequences would then be topologically identical, and hence "similar" to the human eye (see Figure 4). The scheme offered a ready explanation for figure-ground reversals where the same image contour "looked different" depending upon which side of the curve was taken as figure (see Figure 5 and cover). The explanation was simple; a reversal of figure-ground caused maxima of curvature to become minima, thus switching the location of the part boundaries along the curve (and hence also the sign of Gaussian curvature and the intrinsic 3D shapes).

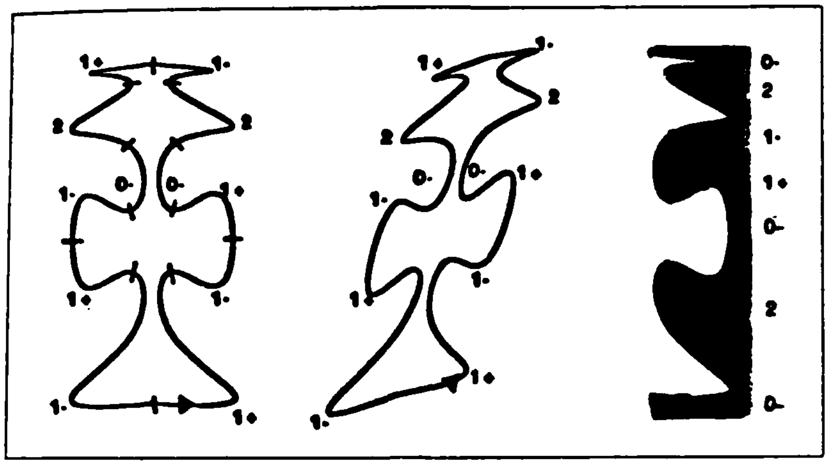


Figure 4 Skewed symmetry is obvious in the codon string because half the sequence is reversed, ignoring the sign of the codon (left frame). Figure-ground reversal changes the codon string because maxima and minima of curvature are exchanged, providing a simple explanation for Rubin's face-vase illusion. (Adapted from Hoffman, 1983.)

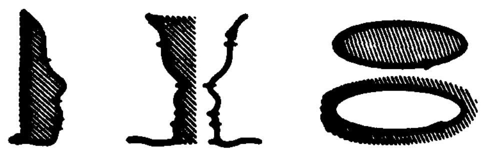


Figure 5 Partitioning plane curves into its parts. Extrema of negative curvature are indicated by slashes. Arrows indicate direction of traversal of curve. The figure is taken to be to the left of the direction of traversal.

A sequence of codons is a very abstract description of a plane curve. This has advantages as a first index into shapes (see Figure 6), but obvious disadvantages when metrical or relational information needs to be specified. Recently, Ullman (1986) and Huttenlocher & Ullman (1987) have proposed a 2D shape recognition scheme that supplements the curvature primitives with relational information. In this scheme, Ullman uses the fact that rigid 3D objects do not

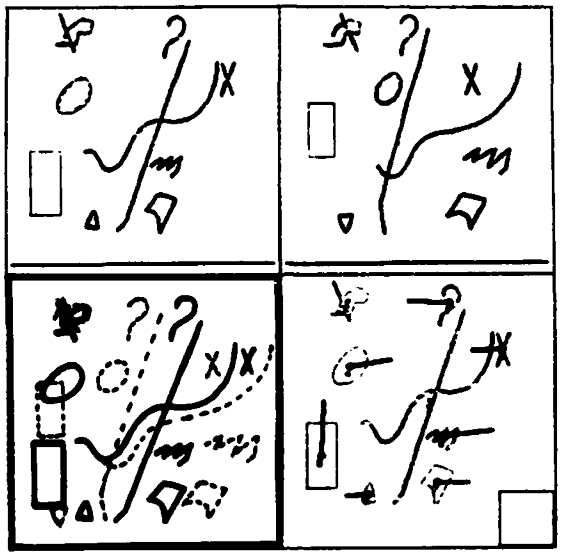


Figure 6 A sloppy stereogram, where images presented to each eye (upper two panels) include shapes that underwent distortions, changes in size and articulations. Matching by shape using codons is robust to these distortions, just as matching to memory must be for recognition. (From Richards, 1988.)

change their appearance in arbitrary ways as they are rotated, and hence the set of transformations from object to image is restricted. Indeed, only three distinct feature points are needed to "undo" the unknown rotation, translation and scale transformations, given the object model, thus permitting unique object-to-model alignment (for rigid objects).

3.0 Processes and Axes

A second major approach to representing 2D shapes is process-based descriptions. Again, curvature and curvature extrema play a key role (Leyton, 1987, 1988, 1989; Milios, 1989). The argument follows from the notion that when shapes are created by either internal or external forces, then these forces are maximal at a point, not a region, and consequently create a curvature extremum on the surface of the shape.[1] As the force is applied over time, this extremum will sweep out a trajectory in space. With rather simple but plausible assumptions, Leyton argues that this trajectory will correspond to the axis of symmetry of the shape. Hence the symmetry axes of a shape reflect the process history of that shape. The inference from curvature extrema to processes thus entails two stages:

$$\text{curvature extrema} \rightarrow \text{symmetry axes} \rightarrow \text{processes.}$$

As we hinted above, the intermediate role of the symmetry axes in the process description is crucial. Two kinds of curvature computations enter here. First, as Leyton proves with his Symmetry Curvature Duality Theorem (1987), each curvature *extrema* corresponds to the termination of a unique differential axis of symmetry. Second, these axes of symmetry are defined by pairs of tangent vectors which are mirror symmetric on circles—a construction we will use later to compute curvature. This construction is illustrated in Figure 7. The differential symmetry axis of two segments of a curve is simply the locus of the midpoint of

[1]More properly, the application of a point force to an (elliptical) surface patch will create two extremum—first an hyperbolic saddle followed by the appearance of Leyton's elliptic extremum.

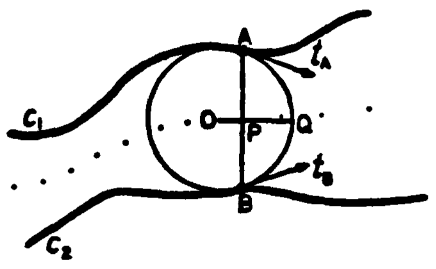


Figure 7 Two curves c_1 and c_2 that have symmetric tangent vectors at A and B. The SAT-axis is the loci of circle centers (represented by the dots shown). The SLS-axis and PISA-axis would be the locus of points P and Q, respectively.

the *circles tangent to the curve* (SAT) or, alternately, some other fixed point on the circle or one of its chords (Blum, 1973; Pizer et al., 1987).

The obvious advantage of including axes in shape representations is that the size orientation and relations between parts can be specified more fully (Marr & Nishihara, 1978). A sequence of codons alone lacks this information. However, as explored rather thoroughly by Leyton (1988), there is a close tie between the codon and process or axial-based descriptions. We shall return to this briefly later.

4.0 Computing 2D Image Curvature

Present algorithms for computing curvature along an image contour can be divided into three categories, (1) the differentiation of an edge list of the contour; (2) the construction of circles tangent to two points along the curve (cocircularity); and (3) symbolic grouping of certain image tokens. Our laboratory has stressed the first approach (although see Saund, 1988 for work on the latter).

4.1 Edge List or Contour Differentiation

In machine vision, once a blob or contour is isolated, its edge list can be numerically differentiated twice to yield curvature (Dawson & Treese, 1984; Richards et al., 1986). Two scale issues arise in this process. First, the blob or contour is customarily found using the Laplacian of Gaussian or some similar two-dimensional smoothing function whose scale is chosen rather arbitrarily. Second, the resulting edge list itself must be smoothed when constructing tangents to the curve, and a scale must also be set for the curvature calculation, which customarily entails simply running an "edge" or first derivative operator over a plot of tangent orientation versus edge position. Alternatively, all of these operations can be done locally using two dimensional masks (Koenderink & Richards, 1988)—but again the spatial scale of the masks must be chosen in advance. To represent the complete spatial structure of an image object, the computations must be performed over a range of scales (Witkin, 1983, 1984; Koenderink, 1984; Yuille & Poggio, 1984). Thus, curvature extrema computed across scales yield a rather complex data structure. How can this data structure be simplified?

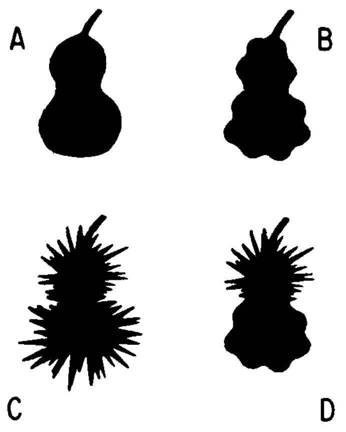


Figure 8 Pears with different surface textures.

Figure 8 illustrates the problem. Let the silhouette of a pear-shaped object (A) be modified to have either a sinusiodal ripple (B), prickles (C) or a composition of prickles and ripples (D). In spite of these modifications of the silhouette, the underlying pear shape is still apparent to us. So is its part structure. This suggests that the coarser scale part boundaries of (A) are still recoverable even when corrupted by fine textured prickles (C). Most algorithms proceed to recover the coarse scale structure of the corrupted silhouette (C) by blurring (as in a Laplacian Pyramid). There are two disadvantages to this rather obvious scheme: (1) the resultant data structure is exceedingly complex: because it carries many 2D descriptions of the image, we are required to compute curvature over and over again for each of these blurred versions of the image—each for a different scale curvature operator! (2) Surprisingly, sometimes these blurred

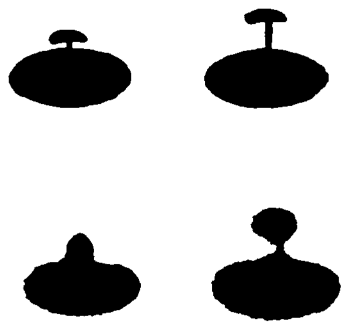


Figure 9 When the top two shapes are blurred, then the handle either merges with the ellipse to create a bump (left) or dissociates itself to create a separate smaller ellipse (right).

versions of the original image can destroy or create new structures, as illustrated by Figure 9. If this figure is blurred, the "handle" will break off in the right-hand version to create two knobs, whereas in the left, the handle merges into a bump on an ellipse. Neither result properly describes this object and its parts as we see it.

4.2 Fine to Coarse Strategy

As an alternate approach, we propose computing tangent orientations along the silhouette only at the finest 2D scale, and then blurring (or grouping) these fine-scale tokens by using larger and larger curvature operators. This approach greatly simplifies the data structure, because only one scale space is constructed, not a multiplicity of scale spaces. The result resembles Witkin's scale-space, but in the curvature domain. Figure 10 shows this method applied to the pears (A)

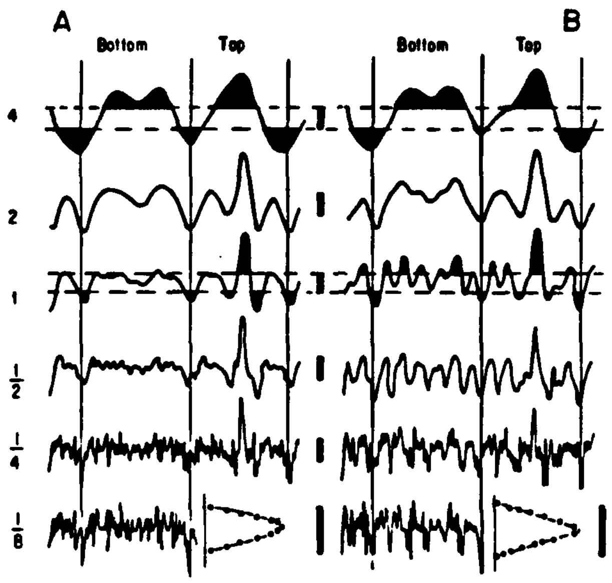


Figure 10 Curvature-space and level-crossing plots for *A*, the smooth pear and *B*, the rippled pear shown in Figure 8. The ordinate shows the relative widths of the curvature operator; positions on the abscissa correspond to positions on the silhouette. All edge lists are normalised to the same length. The two major part boundaries are indicated by the vertical lines through the significant extrema of negative curvature. The vertical bars between the two graphs indicate the region within which extrema are not marked, unless a plateau appears in the level-crossing histogram. These bars correspond to 50% of the range of the filter width indicated by the level-crossing histograms shown as insets at the bottom. These graphs show only the first pass at part decomposition.

and (B). An edge list is created for the silhouette, and tangents are computed at the finest scale for each point on the edge of the silhouette. A curvature operator (1-D edge mask) is then run across the plot of tangent orientation versus edge position to obtain the data of Figure 10. Each curve in this figure shows the output of one scale curvature mask (size labelled at the left) versus position in the edge list. At the finest scale curvature-mask (eg. 1/8), the result is essentially white noise. However, as the size of the curvature mask is increased, more and more structure appears in this kind of scale-space plot. For example, at scale (1/2) the sinusoidal ripples in pear (B) become apparent. And finally, at the coarsest scale used (A), the underlying part structure is obvious, with both pears (A) and (B) yielding similar graphs. [(C) also exhibits this but (D) will not.] Thus, for some silhouettes with homogeneous texture, it appears that smoothing over a fine-scale tangent distribution can be equivalent to blurring the 2D image to remove fine detail. Under these conditions it is not necessary to carry a multiplicity of blurred versions of the original image, with the operations of Figure 10 applied to each. Generally, the finest scale image suffices. (Note, however, that for Figure 8-D, some local normalizations of the texture must be included—see Mokhtarian & Mackworth, 1986; Richards et al., 1986; Saund, 1988.)

4.3 Biological Support

Our fine-to-coarse proposal runs counter to current trends in computer and machine vision. Although a recent study by Saund (1988) shows how the grouping of fine scale tokens can capture coarse-scale structure, additional supporting evidence for this unorthodox approach is desirable. Here we appeal to evidence

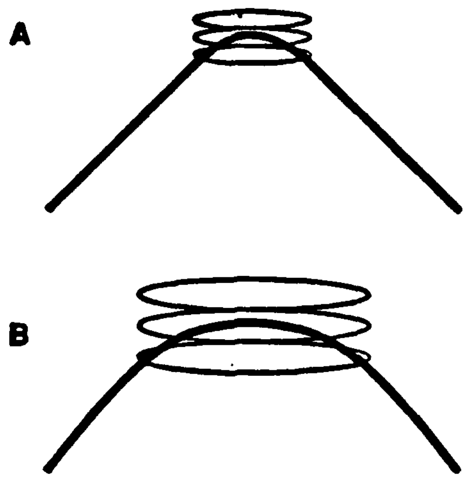


Figure 11 Small masks would be expected to underlie the computation of high curvatures (A), whereas larger masks would seem to be required for computing low curvatures (B).

that one of the most powerful vision processors—our own—seems to use this fine-to-coarse strategy [see Richards (1988) for a stereo example].

Recently, Wilson & Richards (1989) estimated the size of curvature operators using a discrimination paradigm. The idea is quite simple, as illustrated in Figure 11. For high curvatures, small oriented edge or bar masks will be needed. For lower curvatures, the obvious first choice is to simply magnify these same operators. Thus, the expectation is that large scale (tangent) masks will be used for low curvatures, whereas fine scale (tangent) masks compute high curvature.

Contrary to our expectations, fine scale masks are used in the discriminations of both high and low curvatures. Coarse masks are not used to perform discriminations of low curvature. This conclusion is reached from two observations:

(1) Blurring the image contour greatly impaired low curvature discrimination (as well as high curvature discrimination)—contrary to what one would expect if larger, coarser masks are used to compute low curvatures.

(2) Band limiting low curvature contours so they were visible only to the finer channels did not impair curvature discrimination—again suggesting that the finer scale channels were responsible for the low curvature discrimination.

A final, important result, was a demonstration that the fine scale (tangent) operators were capable of computing low curvature provided a neighborhood scheme was used, rather than local differentiation as described in an earlier section. This second curvature mechanism is described below.

4.4 Co-circularity and Curvature

Consider the arc illustrated in Figure 12, with local coordinate frames at points A and B and with the x axis set by the tangent to the contour. Then two tangents x_a and x_b are defined as cocircular if and only if there is a circle to which they are both tangent. Note that by this definition the tangents make equal but opposite angles ϕ to the line joining the points of tangency. Thus cocircularity

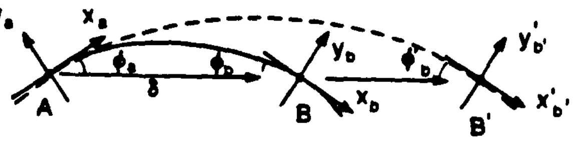


Figure 12 Local coordinate frames at A, B, and B' satisfy the cocircularity constraint when $\phi_a = -\phi_b$. (From Koenderink & Richards, 1988.)

defines a symmetry relation between the tangents and relates several schemes for recovering shapes or their developmental processes (Blum, 1973, 1978; Brady & Asada, 1984; Leyton, 1988). Referring to Figure 12, we see that this bilocal operator for curvature can be defined by the amount of rotation 2ϕ of the local coordinate frame over the displacement δ, namely $\kappa_{cocircle} = 2\phi/\delta$. Parent & Zucker (1987) use this method quite successfully to infer continuity of segments of broken lines or of intersecting contours. It also appears to be the scheme used by the human visual system for curvatures below 10 deg^{-1}, according to the Wilson & Richards data.

The advantages of this second curvature mechanism are clear. If only fine-scale tangents are constructed along a curve, then an exceedingly small angular resolution would be required to compute curvature from adjacent tangents on a curve. However, as the distance between the two tangents increases, the angular resolution required decreases.

A further advantage of two curvature mechanisms is that each really addresses a different problem. Significant part boundaries usually create acute curvature extrema, such as "corners". The local differentiation operator (or its 2D equivalent) is ideally suited for this task, and could easily appear as a parallel computation throughout a (foveal) region. The bilocal or low curvature, co-circular mechanism requires more machinery. It is most useful for testing for curve or contour continuity, either when two curves intersect or when a contour is broken. Generally these are low curvature computations that need to take place at points where high curvature is present, such as at the intersection of two curves where "corners" are created, or at "T" junctions. Thus, a parallel set of high curvature operators could identify regions where the more complex machinery of co-circularity could be profitably applied.

5.0 Inferring 3D Curvature

We now have an image-based data structure that describes a silhouette or image curve in terms of curvature extrema or magnitude. Our next task is to infer the 3D shape from the 2D image contour. As illustrated in Figure 13, our implementation takes in an image, identifies blobs, and then calculates the 2D shape of the blobs in terms of a sequence of codons (upper right). The data can be presented either as a string of curvature versus edge position as in Figure 10, or pictorially for use in alignment schemes as in Figure 13. Here, Snoopy's head has been given symbollic codon labels (1^+, 1^-, 2) that represent a description of

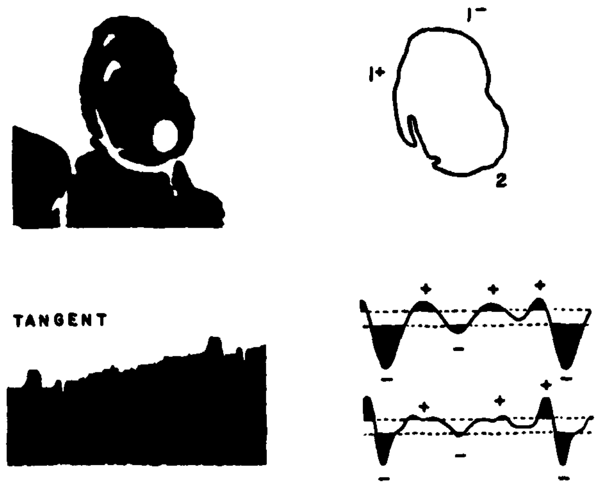


Figure 13 Codon labeling for Snoopy's head, shown as a blob in the upper left-hand panel at one scale of a binary pyramid. Lower left, tangent angle versus position along the contour; lower right, coarsest scale of the curvature space, showing marked curvature extrema; upper right, resultant codon labeling.

this shape at one scale of the curvature computation. As previously mentioned, a sequence of codons provides a complete description of a curve in terms of very primitive elements defined by singularities of curvature (i.e. maxima, minima, and zeros). Because the set is complete at this level of description, we can use the codons to enumerate all possible silhouettes that we may encounter in our images. Because of constraints on joining sequences of codons, there are only 21 possible shapes of smooth objects having four or less codons. Some of these are shown in Figure 14. These shapes will serve as our set of 2D "silhouettes" whose 3D structure we wish to recover. They represent the classes of topologically different outlines that can be constructed from up to four smooth closed codon

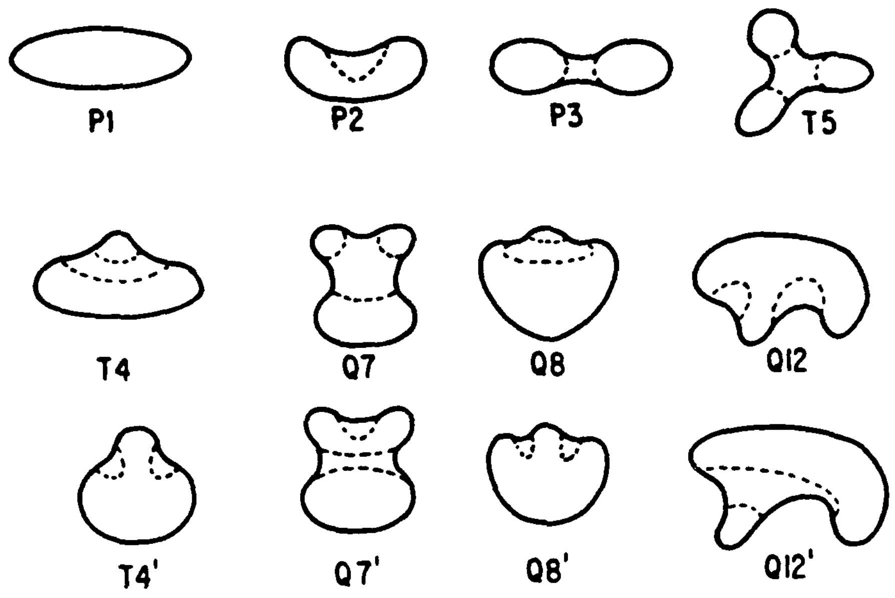


Figure 14 Different classes of the outlines to be interpreted. Dashed lines are the preferred flexional (parabolic) loci. Primes indicate alternative interpretations.

strings. Although this set may appear limited, it is easily extended. It will become clear that our interpretation method will still handle the extended set.

5.1 Choosing a 3D Representation

To begin, we need a means of describing the shape of a 3D surface—its undulations, protrusions, folds, etc. We choose for this purpose an intrinsic property of 3D surfaces, namely Gaussian curvature.

At any point on a smooth (non-planar) surface, there is a direction where the surface curves the most and another where the surface curves the least. These two directions are the directions of principal curvature, and they are always perpendicular (Hilbert & Cohn-Vossen, 1952). The Gaussian curvature is simply the product of the (signed) magnitude of these two curvatures. Of particular interest to us here is simply the sign of the Gaussian curvature, which permits a qualitative description of the topology of the surface in terms of four basic types of "parts": an ellipse, a saddle, a cylinder, or a plane. If the signs of both principal curvatures are identical, the surface patch is elliptical and the Gaussian curvature is positive; when the principal curvatures have opposite signs, the region is a "saddle" and the Gaussian curvature is negative (see Figure 15). A cylinder has one principal curvature equal to zero; for the plane all curvatures are zero and in both these cases the Gaussian curvature is therefore also zero. An ellipse may now be represented as a 3D surface everywhere having positive Gaussian curvature. (Our scheme will thus not distinguish between an ellipse and other totally convex shapes, such as an ovoid.) A 3D dumbbell would be described as two elliptical protrusions of positive Gaussian curvature joined by a hyperbolic region of negative Gaussian curvature. A 3D peanut is a hyperbolic region lying within an ellipsoid of positive Gaussian curvature. Note that our description of 3D shapes in terms of Gaussian curvature captures only the general topology of the shape. Hence a dumbbell and a pear-shaped object have similar

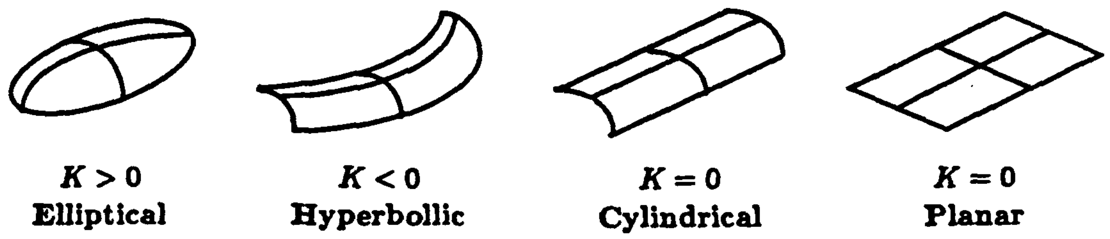


Figure 15 The basic types of surfaces which will be used to describe 3D shapes.

descriptions. To capture these distinctions, metric information can be added at a later stage.

5.2 Inferring Gaussian Curvature from Silhouettes

Consider now the pear-shaped silhouette of Figure 14. Our 3D interpretation is also that of a pear—as if this silhouette were simply a surface of revolution. Why is such an inference justified?

A 3D pear is simply two elliptical protrusions joined by a hyperbolic saddle. The silhouette gives us this information because of the following theorem proved by Koenderink & van Doorn (1982):

C1: The sign of Gaussian curvature of points on the 3D surface that project into the silhouette is the same as the sign of curvature of those projections.

This theorem assures us that the Gaussian curvature of the 3D shape is positive at points on the surface that project into regions of positive curvature on the silhouette. Hence the elliptical outline is the projection of at least a 3D elliptical "ribbon". Similarly, both the peanut and pear (or dumbbell) outlines of Figure 14 require that the corresponding 3D shapes have hyperbolic (saddle) regions of negative Gaussian curvature within a region (or two) of positive Gaussian curvature. Note that this theorem also implies that dents, which have positive Gaussian curvature but are concavities in the surface will never appear in "generic" silhouettes.

At this point, one might be misled to the false conclusion that our problem of inferring the general topology of 3D shapes from silhouettes is solved. However, what about the back side of the silhouette? In principle, an infinity of possible bumps and dents could occur, leaving open unlimited possibilities for the "real" 3D shape. Clearly our preferred, immediate 3D interpretation has assumed that our view is such that more of the bumps or dents of the object are occluded or invisible. In other words all the "part" structure of the object is visible. To capture this motion, we propose the following interpretation rule:

C2: Do not propose undulations of the 3D surface without evidence for such.

The above rule is an extension of the general position restriction, which requires that the view of an object is not a special one and is stable under perturbation. Further evidence supporting this interpretation rule has recently been provided by Warrington (1986), who shows that the appearance of "part" structure in a silhouette is critical to recognition: additional parts are not inferred until they become visible in the silhouette.

Surprisingly, even with the two above constraints we still can not force a unique 3D description (in terms of allowable undulations and protrusions). Consider shape Q7. Is this shape a torso with two "bumps" on top, or is it a dumbbell with a furrow-shaped saddle in one end? To characterize these differences more clearly, we can enclose the regions of positive (and negative) Gaussian curvature as illustrated by the dashed lines in Figure 14. These lines which separate the regions of positive and negative curvature obviously must have zero Gaussian curvature—they are locally cylindrical patches. Clearly these lines of zero Gaussian curvature—the so-called flexional or parabolic lines—must go through the inflection points on the silhouette. Again, Koenderink & van Doorn (1987) have proven an important theorem about the behavior of these lines:

> C3: For smooth generic surfaces, the flexional parabolic lines of zero Gaussian curvature are closed and non-intersecting.

Thus, for a simple pear-shaped object with four inflexions on the silhouette, we are allowed only two 3D decompositions as illustrated in Figure 16. Generally, as proven by Beusmans et al. (1987), there will be $\left[{n \atop n/2}\right]/(n/2+1)$ possible legal generic interpretations of a silhouette having N inflections. Thus shape Q7

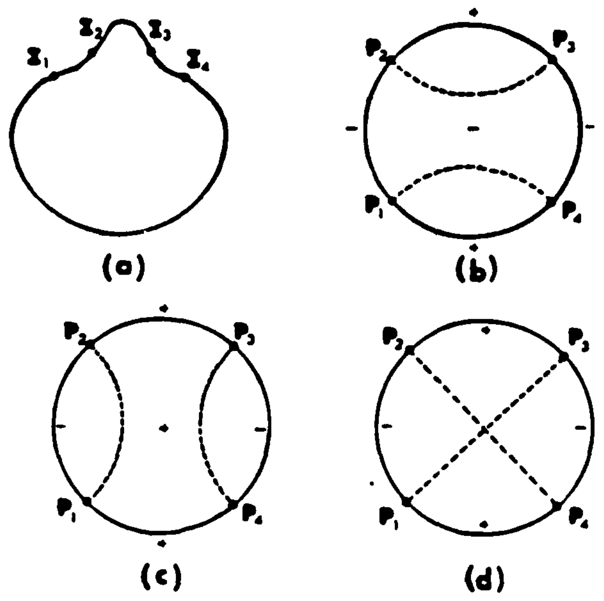


Figure 16 (a) A contour having four inflection points (I_1, I_2, I_3, I_4). The corresponding four parabolic points (P_1, P_2, P_3, P_4) on the fold can be paired generically (b), (c) or nongenerically (d). (From Beusmans et al., 1987.)

has 14 possible legal decompositions, yet only the two shown in Figure 14 are readily visualized. This suggests further constraints or assumptions are imposed upon our interpretations of silhouettes. As discussed elsewhere, one of the key interpretation rules appears to be an extension of the general position rule (C2) mentioned earlier:

C4: Without evidence to the contrary, pick the most general position 3D interpretation, namely that 3D shape which preserves the signs of Gaussian curvature of the silhouette over the widest range of viewpoints.

This rule favors surfaces of revolution, and excludes local furrows (saddles) whenever possible. Thus, the preferred interpretation for the dumbbell or pear-shaped object is just that, and not something created by two separate saddle-shaped furrows. This rule reduces the preferred interpretations of our given initial set of topologically different shapes shown in Figure 14 to those illustrated. Thus, for any given 2D image silhouette, we obtain a very restricted set of preferred 3D interpretations from the infinity of possibilities.

6.0 Parts, Processes and Perceivers

Now consider the amoeba-like shape shown in Figure 17. Although we have rules for describing this silhouette as a 3D shape in terms of regions of positive and negative Gaussian curvature, a more desirable description would be in terms of a part-based structure. After all —we don't describe a human body as a collection of elliptical and hyperbolic patches, but rather as a torso with a head, neck, arms, legs, etc. How can we convert a description based on Gaussian curvature into the more useful part-based representation? Clearly we know where the part boundaries are—these are the extrema of negative curvature. But given one such concavity, to which should it be paired? We will consider two approaches to this problem: (1) the process-history description (Leyton, 1988) and (2) the Gaussian curvature description (Richards, Koenderink & Hoffman, 1987). Each

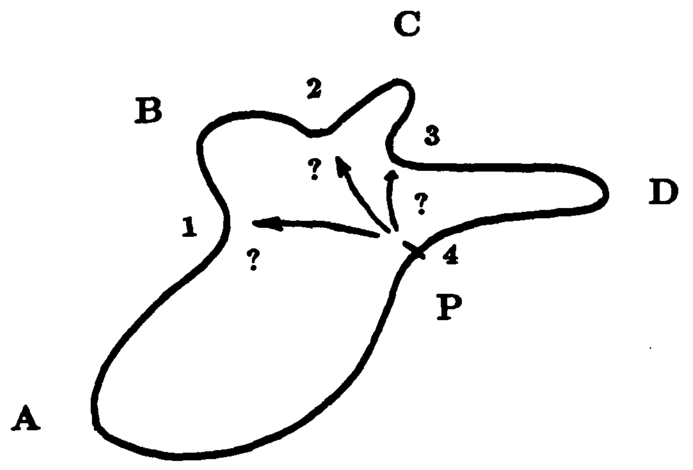


Figure 17 Point *P* is a part-boundary, being an extremum of negative curvature. To which other extremum should the boundary be paired?

method produces fourteen different part-based descriptions of this simple amoeboid shape. A few of each are illustrated in Figure 18. The problem is to find a set of rules which will pick out the preferred interpretations.

A major step in solving this type of problem has been made recently by Jepson & Richards (1989). The advance was showing that a perception can be regarded as an interpretation associated with a locally maximal node in a lattice of possible interpretations of the sense data. To construct such a lattice, it is necessary to invoke models of the world which are not always correct. Generally, one or more of the model elements must be given up or "faulted" when constructing a possible interpretation.

For example, the possible interpretations of the amoeboid shape shown in Figure 18 may be ordered by preferring shapes which satisfy the following:

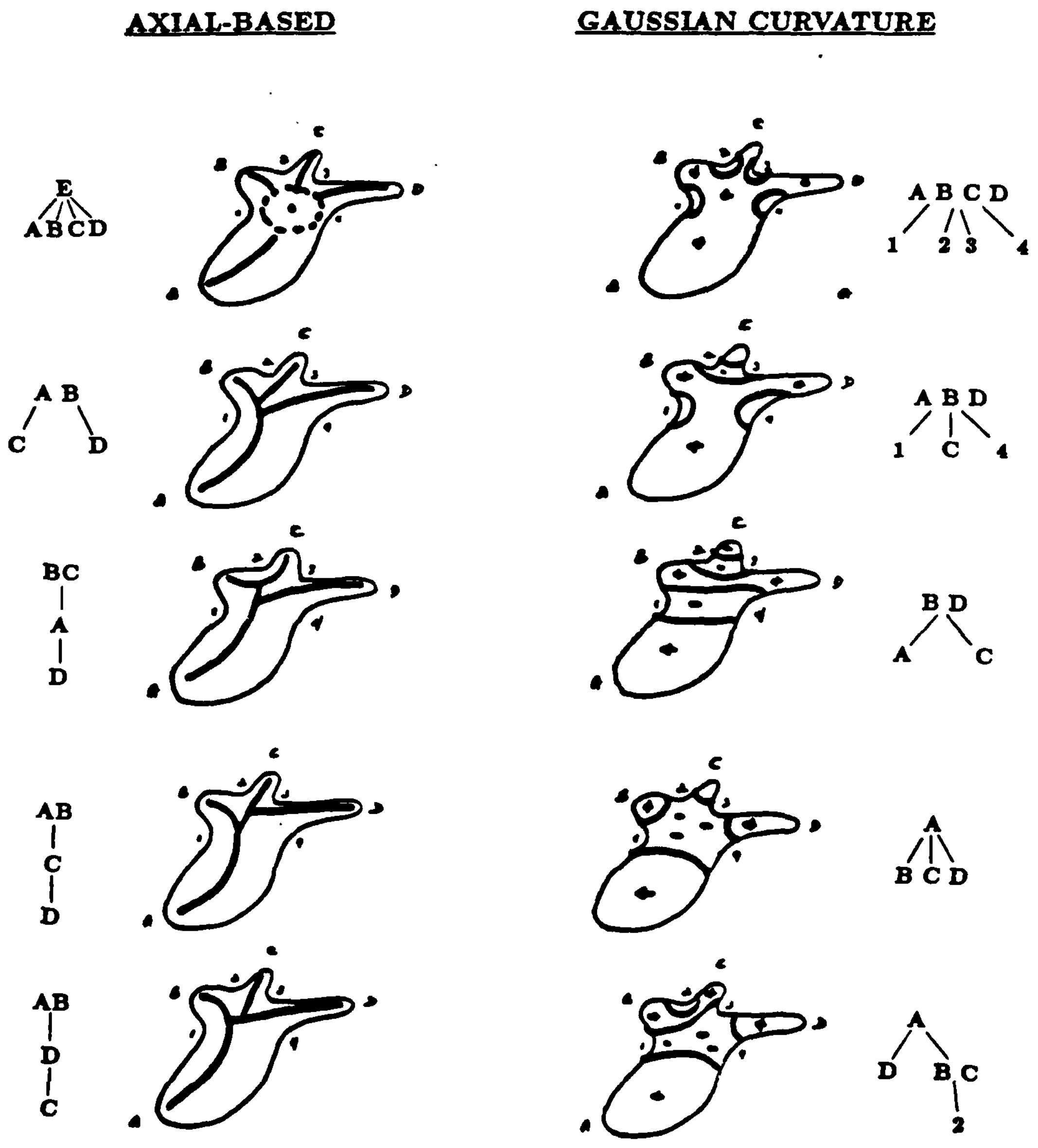


Figure 18 Possible part-based descriptions for an amoeboid shape using axial-based decomposition (left) or one based on Gaussian curvature (right). The trees at the left and right margins suggest the process history.

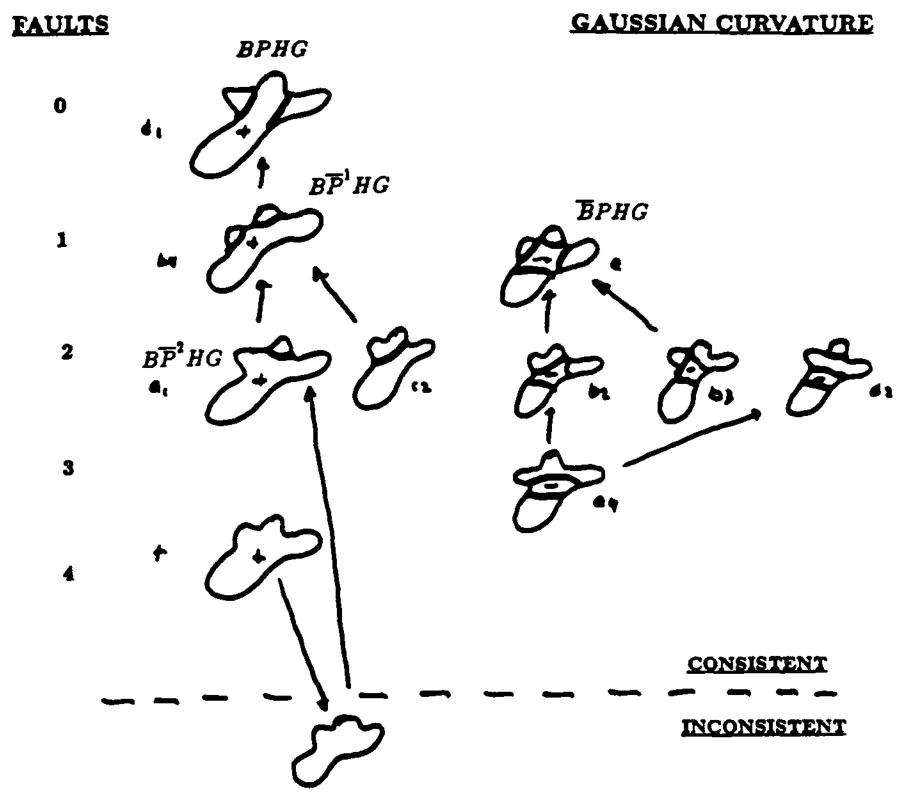


Figure 19 A partial ordering of part-based interpretations for an amoeboid shape, using Gaussian curvature primitives. Model premises B, P, H, G are given in the text. The highest node in the lattice ($BPHG$) requires giving up (or faulting) none of the preferred models. There is also another maximal node ($\overline{B}PHG$) which faults one model premise (B).

B:	The Object Body is the largest part.
P:	Parts are convex.
H:	Parts are separated by hyperbollic regions.
G:	The Body contains the center of mass.

A portion of the resultant lattice for interpretations based on Gaussian curvature is shown in Figure 19. There are two locally maximal nodes ($BPHG$) and

($\overline{B}PHG$). These interpretations correspond respectively to an ovoid with two symmetric protrusions ($BPHG$) and a hyperbolic body with four protrustions ($\overline{B}PHG$). As indicated by the bar, one rule was violated for the latter interpretation. But still more violations would have occured if other submaximal interpretations were accepted, as indicated by the superscripts. Similarly, the process-based descriptions can be given a partial ordering using slightly different rules which apply to axes. This lattice is different from that shown in Figure 19, although one of the locally maximal nodes is identical to $BPHG$, and hence would yield a perception with the same part-based description. Additional psychophysical studies of lattices based on different representations are in progress.

Finally, we now have some insight into how to proceed in our analysis of occluded shapes. If our interpretation rules are context-sensitive—as they should be—then the structure of our lattice of interpretations can be drastically altered by neighboring structures (in space or time). Thus, in Figure 20A the context suggests "disc-like" models, and the occluded shapes are so interpreted. But in the presence of a "peanut" world (Figure 20B), the occluded shape is now more readily seen as a peanut. Similarly, if texture is added to one of the occluded halves, but not the other, then the percept that there is only one object would require violating the notion that an object should be uniformly colored and textured. Hence two different objects will be inferred. The "Perceiver Framework"

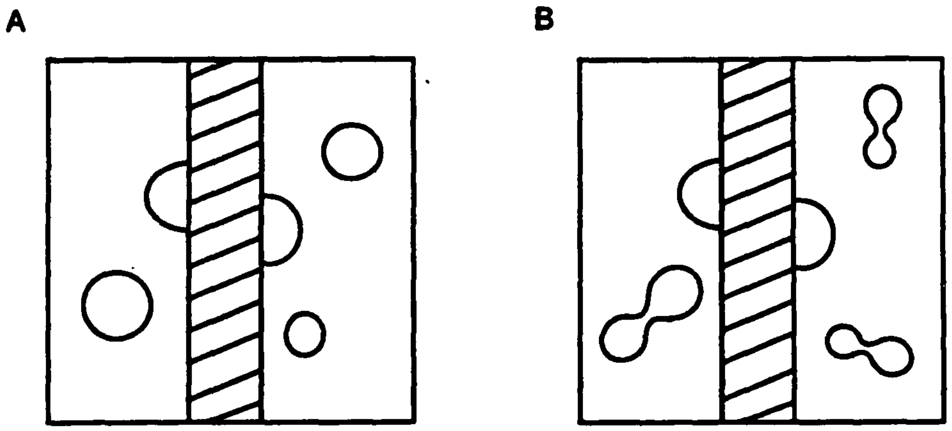


Figure 20 In window (A), the pole appears to occlude two separate discs. However, in window (B), a single dumbbell-shaped object seems more likely.

is a vehicle for making such inferences quite rigorous, and is likely to drive a good portion of our future research.

7.0 Summary

We have roamed over a rather broad territory in order to illustrate the advantages and difficulties encountered in using curvature as a basis for inferring the structure of smooth 3D shapes. Thanks to several implementations, we know that the computation of image curvature is feasible. Given this 2D description, assigning 3D Gaussian curvature is straightforward. The difficulty is that at present our programs are not powerful enough to choose among the possible 3D interpretations, especially in the presence of occlusions. Recent work on fault lattices offers one promising approach, however. Again, we expect to rely heavily upon hints from human vision to guide us in how the 3D shape interpretations might be made from 2D image contours, or silhouettes. An important step would be to embody these newer ideas in a working machine program, just as has been done for the computation of image curvature. Another would be to extend the analysis from smooth objects to include those with fractal-like structures.

8.0 References and Bibliography

***These publications were supported by this contract.**

Attneave, F. (1954) Some informational aspects of visual perception. *Psychological Review*, 61:183–193.

Banchoff, T., Gaffrey, T. & McCrory, C. (1982) *Cusps of Gauss Mappings*. Pitman, Boston.

Bennett, J.B. & Hoffman, D. (1985) Shape decomposition via transversality. In *Image Understanding 1985–86* W. Richards & S. Ullman (eds.), Ablex Publishing, Norwood, N.J.

Beusmans, J.M.H., Hoffman, D.D. & Bennett, B.M. (1987) Description of solid shape and its inference from occluding contours. *Jrl. Opt. Soc. Am. A*, 4:1155–1167.

Biederman, I. (1986) Recognition by components: a theory of image interpretation. In *Human and Machine Vision II*, A. Rosenfeld (ed.), pp. 13–57.

Blum, H. (1973) Biological shape and visual science (part 1). *Jrl. Theoretical Biology*, 38:205–287.

Blum, H. & Nagel, R.N. (1978) Shape description using weighted symmetric axis features. *Pattern Recognition*, 10:167–180.

*Bobick, A. (1984) Grouping visual targets. Abstract. Perceptual Organizations Workshop, Pajaro Dunes, California.

*Bobick, A. (1984) Using mirror reflections to recover shape. Abstract. WH2, Optical Society of America Annual Meeting.

*Bobick, A. (1987) Natural object categorization. Ph.D. thesis, MIT Dept. of Brain and Cognitive Sciences; and MIT AI Lab. Tech. Report 1001.

*Bobick, A. & Richards, W. (1986) Classifying objects from visual information. *MIT AI Memo 879.*

Brady, M. & Asada, H. (1984) Smoothed local symmetries and their implementation. *Int. Jrl. Robtics Research*, 3:36–61.

Bruce, J.W. & Giblin, P.J. (1985) Outlines and their duals. *Proc. Lond. Math. Soc.*, 50:552–570.

Burt, J.B. & Adelson, E.H. (1983) The Laplacian Pyramid as a compact image code. *IEEE Trans. on Comm.*, Vol. COM-31, No. 4, pp. 532–540.

Cayley, A. (1859) On contour and slope lines. *London Edinburgh Dublin Philos. Mag. J. Sci.*, 18(120):264–268.

*Dawson, B. (1987) Introduction to image processing algorithms. *Byte*, #3, 12:169–186.

*Dawson, B. (1987) Recognizing objects using curvature. *ESD*, 17:96–100.

*Dawson, B. (1988) Focusing on image enhancement. *ESD*, #3, 18:83–86.

*Dawson, B. (1989) Hardware and codon algorithms for parts matching. *Electronic Imaging '89 West*. Boston: BIS/CAP, 1:363–368.

*Dawson, B. (1989) Using codons to describe object outlines. *IEEE International Conference on Image Processing*, 2:827–831.

*Dawson, B. (1989) Changing perceptions of reality. *BYTE*, 14:293–304.

*Dawson, B. (1990) Improving the Sobel Edge Operator. *Electronic Imaging '90 West*. To be published Feb. 1990.

*Dawson, B. & Hallinan, P.W. (1987) TOOLKIT Image Processing Software, Version 1.0. (Available MIT, Natural Computation Group, 79 Amherst St., Cambridge, MA 02139).

*Dawson, B. & Treese, W. (1984) Computing curvature from images. *Proceedings SPIE*, 504:175–182.

*Dawson, B. & Treese, W. (1985) Locating objects in a complex image. *SPIE Conference on Image Processing. Proceedings SPIE*, 534:185–192.

Fischler, M.A. & Bolles, R.C. (1983) Perceptual organization and curve partitioning. In *Proc. Image Understanding Workshop*, pp. 224–322.

*Gilson, W. (1984) Idiot's guide to OZ (a manual for text editing). Natural Computation Group, MIT.

Hamscher, W. (1986) Codon constraints on 2D cusps. MIT Artificial Intelligence Laboratory. Personal communication.

Hilbert, D. & Cohn-Vossen, S. (1952) *Geometry and the Imagination*. (Translated by P. Nemeny.) Chelsea, New York.

*Hildreth, E.C. (1984) The computation of the velocity field. *Proc. Roy. Soc. Lond. B*, 221:189–220. Also MIT AI Memo 734. Also chapt. 17 in *Natural Computation*, W. Richards (ed.), MIT Press, Cambridge, Mass.

*Hildreth, E.C. (1984) Computations underlying the measurement of visual motion. MIT AI Memo 761.

Hoffman, D.D. (1983) Representing shapes for visual recognition. Ph.D. thesis, MIT Dept. of Brain and Cognitive Sciences.

*Hoffman, D. & Richards, W. (1982) Representing smooth plane curves for recognition: implications for figure-ground reversal. *Proc. National Conference on Artificial Intelligence*, pp. 5 – 8. Also, in slightly augmented form, chapt. 6 in *Natural Computation*, W. Richards (ed.), MIT Press, Cambridge, Mass.

*Hoffman, D.D. & Richards, W.A. (1984) Parts of recognition. *Cognition*, 18:65 – 96 (1984). Also in *Readings in Computer Vision*, M. Fischler and O. Firschien, Morgan, Kaufmann, Los Altos, 1987.

Huttenlocher, D.P. & Ullman, S. (1987) Object recognition using alignment. MIT AI Memo 937.

*Jepson, A. & Richards, W. (1989) Perception and Perceivers. Presented May 25 at a meeting on Vision and Three Dimensional Representation, University of Minnesota, Minneapolis, Minnesota. (To appear as Univ. Toronto, Dept. of Computer Science Tech. Report.)

Koenderink, J.J. (1984) The structure of images. *Biol. Cybern.*, 50:363 – 370.

Koenderink, J.J. (1987) The internal representation for solid shape based on the topological properties of the apparent contour. In *Image Understanding 1985 – 86* W. Richards & S. Ullman (eds.), Ablex Publishing, Norwood, N.J.

Koenderink, J.J. & van Doorn, A.J. (1976) The singularities of visual mapping. *Biol. Cybernetics*, 24:51 – 59.

Koenderink, J.J. & van Doorn, A.J. (1980) Photometric invariants related to solid shape. *Opta. Acta.*, 27:981 – 996.

Koenderink, J.J. & van Doorn, A.J. (1981) A description of the structure of visual images in terms of an ordered hierarchy of light and dark blobs. *Proc. Second International Visual Psychophysics and Medical Imaging Conference*, New York.

Koenderink, J.J. & van Doorn, A. (1982) The shape of smooth objects and the way contours end. *Perception*, 11:1129 – 1137. Also chapt. 10 in *Natural Computation*, W. Richards (ed.), MIT Press, Cambridge, Mass.

*Koenderink, J.J. & Richards, W. (1988) Two-dimensional curvature operators. *J. Opt. Soc. Amer. A*, 5:1136 – 1141.

*Leyton, M. (1985) Generative system of analyzers. *Computer Vision*, 31:201 – 241.

Leyton, M. (1986) Principles of information structure common to six levels of the human cognitive system. *Information Sciences*, 38:1–120.

*Leyton, M. (1987a) Symmetry-curvature duality. *Computer Vision, Graphics and Image Processing*, 38:327–341.

Leyton, M. (1987b) Nested structures of control: an intuitive view. *Computer Vision, Graphics & Image Processing*, 37:20–35.

*Leyton, M. (1987c) A limitation theorem for the differentiation prototypification of shape. *Jrl. Mathematical Psychology*, 31:307–320.

*Leyton, M. (1988) A process-grammar for shape. *Artif. Intell.*, 34:213–247.

*Leyton, M. (1989) Inferring causal history from shape. *Cognitive Science*, 13:357–387.

Lowe, D.G. (1985) *Perceptual organization and visual recognition.* Kluver Academic Publishers, Boston.

Marr, D. (1982) *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information.* Freeman Press, San Francisco.

Marr, D. & Nishihara, H.K. (1978) Reprsentation and recognition of the spatial organization of three-dimensional shapes. *Proc. Roy. Soc. Lond. B*, 200:269–294.

Maxwell, J.C. (1870) On hills and dales. *London Edinburgh Dublin Philos. Mag. J. Sci.*, 40(269):421–427.

Milios, E.E. (1989) Shape matching using curvature processes. *Computer Graphics & Image Processing*, 46, in press.

Mokhtarian, F. & Mackworth, A. (1986) Scale-based description and recognition of planar curves and two-dimensional shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8:34–43.

*Neumann, Y., Schechtman, G. & Ullman, S. (1990) Self-calibrated collinearity detectors. *Biological Cybernetics*, in press.

Pizer, S.M., Oliver, W. & Bloomberg, S.H. (1987) Hierarchical shape description via the multiresolution of the symmetric axis transform. *IEEE Transactions PAMI*, 9:505–511.

Resnikoff, H.L. (1987) *The Illusion of Reality: Topics in Information Science.* Springer-Verlar, N.Y.

*Richards, W. (1984) Grouping without correspondence. Abstract. WP1, Optical Society Annual Meeting.

*Richards, W. (1985) Structure from stereo. **2**:343 – 349. Also *MIT AI Memo 731* (1983).

*Richards, W. (1988) *Natural Computation.* MIT Press, Cambridge, MA.

*Richards, W. (1988) A cyclopean experiment. *Optics News*, **14**:212.

*Richards, W. & Bobick, A. (1988) Playing twenty questions with nature. In *Computational Processes in Human Vision: An Interdisciplinary Perspective*, Z. Pylyshyn (ed.), Ablex Publishing, Norwood, N.J.

*Richards, W. & Hoffman, D.D. (1985) Codon constraints on closed 2D shapes. *Computer Vision, Graphics, and Image Processing*, 31(3):265 – 281. Also in A. Rosenfeld (ed.), *Human and Machine Vision II*, Orlando, Fla:Academic Press, pp. 207 – 223, and MIT AI Memo 769 (1984).

*Richards, W. & Lieberman, H.R. (1985) Correlation between stereo ability and the recovery of structure-from-motion. *Amer. Jrl. Optom. Physiol. Optics*, **62**:111 – 118.

*Richards, W. & Ullman, S. (eds.) (1987) *Image Understanding 1985 – 86*, Ablex Publishing, Norwood, N.J.

*Richards, W., Dawson, B. & Whittington, D. (1986) Encoding contour shape by curvature extrema. *J. Opt. Soc. Amer. A*, **3**:1483 – 1491. Also chapt. 7 in in Natural Computation, W. Richards (ed.), MIT Press, Cambridge, Mass.

*Richards, W.A., Koenderink, J.J. & Hoffman, D.D. (1987) Inferring three-dimensional shapes from two-dimensional silhouettes. *J. Opt. Soc. Amer. A*, **4**:1168 – 1175. Also *MIT AI Memo 840* (1985).

*Richards, W., Nishihara, H.K. & Dawson, B. (1988) CARTOON: A biologically motivated edge detection algorithm. Also chapt. 4 in *Natural Computation*, W. Richards (ed.), MIT Press, Cambridge, Mass. Also MIT AI Memo 668 (1982).

Rosenfeld, A. (ed.) (1984) *Multiresolution image processing and analysis.* Springer-Verlag, Berlin.

*Rubin, J. (1984) Telling actors from objects. Abstract. WP4, Optical Society of America Annual Meeting.

*Rubin, J.M. & Richards, W.A. (1987) Spectral categorization of materials, *Image Understanding 1985 – 86*, W. Richards & S. Ullman, eds., Ablex Publishing, Norwood, pp. 20 – 44.

Saund, E. (1988) The role of knowledge in visual shape representation. Ph.D. thesis, Dept. of Brain and Cognitive Sciences, MIT; also MIT AI Lab. Tech. Report 1092.

*Spalding, M. & Dawson, B. (1986) Finding the Titanic. *Byte*, 11:96–110.

*Truvé, S. & Richards, W. (1987) From Waltz to Winston (via the Connection Table). *Proc. First Int. Conf. Comp. Vision*, June 1987.

*Ullman, S. (1984) Visual routines. *Cognition*, **18**:97–159. Also Center of Cognitive Science Occasional Paper 20 (1984).

Ullman, S. (1986) An approach to object recognition: aligning pictorial descriptions. MIT AI Memo 931.

*Ullman, S. & Richards, W. (eds.) (1984) *Image Understanding 1984*, Ablex Publishing, Norwood, N.J.

*Ullman, S & Richards, W. (eds.) (1989) *Image Understanding 1987–88*, (eds.), Ablex Publishing, Norwood, N.J..

Vaina, L.M. & Zlatera, S.D. (1988) The largest convex patches: a boundary based method from obtaining object parts. Boston University Intelligence Systems Lab. Working Paper 88–85.

Warrington, E.K. & James, M. (1986) Visual object recognition in patients with right hemisphere visions: axes or features? *Perception*, **15**:355–366.

Watt, R.J. (1984) Further evidence concerning the analysis of curvature in human foveal vision. *Vision Res.*, **24**:251–253.

Whitney, H. (1955) On singularities fo mappings of Euclidian spaces. I. Mappings of the plane into the plane. *Ann. Math.*, **62**:374–410.

*Wildes, R.P. (1988) Recovering view and world geometry from stereo disparities. *Optics News*, **14**:219.

Wildes, R.P (1989) On interpreting stereo disparity. Ph.D. thesis, MIT Dept. of Brain and Cognitive Sciences. Appears as MIT AI Lab. Tech. Report 1112.

Wilson, H.R. (1985) Discrimination of contour curvature: data and theory. *Jrl. Opt. Soc. Am. A*, **2**:1191–1199.

*Wilson, H.R. & Richards, W. (1988) Curvature and separation discrimination at texture boundaries. *Invest. Ophthal. Vis. Sci. Suppl.*, **29**:408.

*Wilson, H.R. & Richards, W.A. (1989) Mechanisms of contour curvature discrimination. *J. Opt. Soc. Amer. A*, **6**:106–115.

Witkin, A.P. (1983) Scale-space filtering. *Proc. International Joint Conference on Artificial Intelligence*, pp. 1019 ~ 1022. Karlsruhe, Germany.

Witkin, A.P. (1984) Scale-space filtering: a new approach to multi-scale description. In *Image Understanding 1984*, S. Ullman & W. Richards (eds.), Ablex Publishing, Norwood, N.J., 1984

Witkin, A.P., Terzopoulos, D. & Kass, M. (1987) Signal matching through scale space. *International Jrl. Computer Vision,* **2**:133 – 144.

*Yuille, A. & Poggio, T. (1984) Fingerprint theorems. *Proc. Assoc. Artificial Intelligence (AAAI-84).*

9.0 Appendix: User's Guide to TOOLKIT

Integer Operations
Interpolative Zoom
Line Segment
List Windows
Make 1D Fractal
Make 2D Fractal
Make Window
Median Filter
Mouse Block Fill
Mouse Block Read
Mouse Dots
Mouse Drag New Window
Mouse Jagged Line
Mouse Make Window
Mouse Pixel Read
Mouse Polygon
Mouse Seeded fill
Mouse Spline
Mouse Straight Line
OH Curve
Print Picture
Put Picture
Quit
Random Binary Image
Rectangle
Refresh Terminal
Replace Color

Replicative Zoom
Rotate 90
Run Script
Scale Pixels
Script Off
Script On
Set Env
Show Env
Show Title
Snap Picture
Statistics
Subsampling Reduce
Superimpose
Text
Thin
Threshold
To Core
To Screen
Translate Pixels
Trestore
Trim
Tsave
Verify Fractal
Wedge
White Noise
Zero Crossings

1. User's Guide to TOOLKIT

1.1 Introduction

This guide describes the purpose of TOOLKIT, the concepts needed to understand its operation, and the commands needed to use it.

TOOLKIT was developed by:

Benjamin M. Dawson and Peter W. Hallinan
E10 - 120
Dept of Brain and Cognitive Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Work on TOOLKIT was supported in part by AFOSR contract F 49620-83-C-0135.

TOOLKIT is designed to facilitate the application of image processing routines to color, binary, and greyscale images. Simple graphics routines are also included to allow both the construction of artificial images having a particular intensity function (e.g. perfect ramp edges) and the modification of real images.

TOOLKIT is motivated by four goals:

1) To provide an interactive interface usable by both beginners and experts.

2) To exploit the variety of hardware - graphics printers, mice, graphics displays, and video cameras - available to the Natural Computation Group using device-independent code.

3) To consolidate existing programs into an easily usable whole and provide a structure to which future programs can be easily added.

4) To provide a turnkey demonstration system.

1.1.1 Required Hardware

This version of TOOLKIT requires a VT-100 terminal or VT-100 compatible terminal (for split-screen operation). If you attempt to run this version of TOOLKIT on another kind of terminal, you will see junk on the screen. Other versions of TOOLKIT run under windowing systems on workstations such as the SUN.

TOOLKIT can be used without a display device, although not being able to see the images and results makes using it difficult. If a display device is used, it must have at least 8 bits per pixel, and preferably 24 bits per pixel (8 bits each for red, green, and blue). The display device must also have at least 512 by 512 pixels, not all of which need be displayed.

This version of TOOLKIT supports the Ikonas (Adage) 3000 and the Raster Technologies Model one/25 for image display. Images can be acquired with the Ikonas, or from disk files. It is a fairly easy task to change TOOLKIT to support other image display and acquisition devices.

1.2 Entering TOOLKIT

At the Unix prompt type:

```
toolkit [-acei] [script files...]
```

You will see a copyright line in the middle of the screen separating a menu window at the top from an interactive window at the bottom. The TOOLKIT prompt > appears in the interactive window and TOOLKIT awaits your command.

1.2.1 Executable Images

This version of TOOLKIT exists as a set of device dependent executables:

```
/usr/menu/SRC/ms_iknew
/usr/menu/SRC/ms_srt
```

for the Ikonas (Adage) 3000 and the Raster Technologies model one/25.

A UNIX C Shell script file:

```
/usr/local/toolkit
```

selects the correct executable depending on C shell environment variables that are set at login. These variables name the printers, pointing devices (e.g. mice), and display devices that are closest (in space) to the terminal logged into. Thus for a user sitting next to a display connected to the Ikonas, TOOLKIT will invoke the image compiled with the Ikonas specific routines.

1.2.2 The C Shell Environment

The following variables should be set before calling TOOLKIT:

```
ROOM
PRINTER
GPRINTER
DISPLAY
POINTER_TYPE
POINTER_IO
```

They can be set by hand or the following commands may be placed in your .login file:

```
setenv ROOM            `gethardware room`
setenv PRINTER         `gethardware printer`
setenv GPRINTER        `gethardware gprinter`
setenv DISPLAY         `gethardware display`
setenv POINTER_TYPE    `gethardware pointer_type`
setenv POINTER_IO      `gethardware pointer_io`
```

gethardware is a program that accesses the database /etc/hardware to find the value of the key (e.g. room) passed to it.

1.2.3 TOOLKIT Switches

Four switches may be specified on the UNIX command line in order to tell TOOLKIT to:

a   abort if an error occurs while executing a script file argument.

c   create all image windows in core because you will not be using a graphics display.

e echo to the terminal screen command lines found in script files passed as arguments.

i initialize the graphics display upon starting TOOLKIT.

The -i and -c switches are mutually exclusive.

Switches must be prefixed by a minus sign unless two or more switches are concatenated together, in which case only the first need be prefixed by -. The order of the switches with respect to the arguments is unconstrained.

1.2.4 TOOLKIT Arguments

You may specify any number of script files on the command line (up to the limits imposed by UNIX). Wildcards and other such characters are expanded by the C Shell in the normal fashion. Thus:

```
toolkit *.s
```

runs all script files suffixed with ".s" in the working directory.

1.2.5 Startup Script

When starting up, TOOLKIT automatically tries to run the script file:

```
tkinit.s
```

TOOLKIT first looks for tkinit.s in the current working directory. If the script is not found, then TOOLKIT looks in the home directory. If tkinit.s is still not found, TOOLKIT goes on to run any script files passed as arguments.

1.2.6 Initialization

TOOLKIT initializes in the following manner:

1) check arguments and switches for validity
2) create terminal windows
3) instantiate the environment variables
4) create image windows (see section 2.3 on switch -c)
5) open the display device
6) initialize the display device if -i specified (see section 2.3)
7) execute the startup script file tkinit.s
8) execute any script files passed as arguments
9) set up the terminal screen
10) start interactive session

To repeat, TOOLKIT will not automatically initialize the display device, unless you issue the Snap command when using the Ikonas display, in which case the display is initialized before each video buffer is grabbed.

1.3 Exiting TOOLKIT

To exit TOOLKIT permanently, you may:

1) type "quit" or any abbreviation thereof, or
2) type Control-C

Both commands restore the terminal screen to normal, close any open script files, and exit.

To stop TOOLKIT temporarily to do other work in the C shell, type Control-Z. This restores the terminal screen and stops the TOOLKIT process. Upon restarting TOOLKIT, you may use the command Refresh Terminal to redisplay the menu and interactive windows.

1.4 Menus

Menus are lists of commands organized hierarchically by function. The current menu is the menu displayed in the top half of the terminal screen.

There are two special menu commands displayed in every menu:

Main Menu:	Forces the current menu to be the topmost (root) menu.
Up Menu:	Forces the current menu to be the parent of the current menu.

To set the current menu to be any other menu, simply type the name of the menu at the TOOLKIT prompt, e.g.:

```
> System Menu
```

Typing ? or ?? at the TOOLKIT prompt will cause a one line description of the current menu to be printed on the screen.

Typing the name of a menu followed by a :? or :?? will cause a one line description of the requested action to be printed on the screen.

1.5 Image Windows

TOOLKIT manipulates image windows (two dimensional pixel regions). Windows can be located in core memory or on a graphics display. Windows located in core memory DO NOT share pixel regions even if their positions and sizes indicate they overlap. Windows located on a graphics display DO share pixel regions if they overlap. Thus clearing a graphics display window that overlaps a second window will result in the second window being (partially) cleared as well.

The top left corner of both the screen and a window is numbered (0,0).

Windows have six attributes of concern to the user:

1) x position	- x location of top left corner of window in screen coordinates. Unused but still specified for core windows.
2) y position	- y location of top left corner of window in screen coordinates. Unused but still specified for core windows.
3) dx	- width of window in pixels.
4) dy	- height of window in pixels.
5) updated	- specifies whether the window is updated in core or on the screen.
6) tsaved	- specifies (for a window updated on screen) if the window has been temporarily saved in core (a kind of temporary backup).
7) title	- a string associated with the window. It can be displayed only by explicit command.

When TOOLKIT is run using a graphics device, the following windows are automatically created at initialization time:

Name	X	Y	DX	DY	Type	Updated	Tsaved?
g0	0	0	512	512	Grey	on screen	no
g1	0	0	256	256	Grey	on screen	no
g2	256	0	256	256	Grey	on screen	no
g3	0	256	256	256	Grey	on screen	no
g4	256	256	256	256	Grey	on screen	no
c0	0	0	512	512	Color	on screen	no
c1	0	0	256	256	Color	on screen	no
c2	256	0	256	256	Color	on screen	no
c3	0	256	256	256	Color	on screen	no
c4	256	256	256	256	Color	on screen	no
tv	0	0	512	480	Grey	on screen	no

When TOOLKIT is run without an image display device, the following windows are automatically created:

Name	X	Y	DX	DY	Type	Updated	Tsaved?
g0	0	0	512	512	Grey	in core	---
g1	0	0	256	256	Grey	in core	---
g2	256	0	256	256	Grey	in core	---
g3	0	256	256	256	Grey	in core	---
g4	256	256	256	256	Grey	in core	---

Any of the above windows can be deleted; they are not special in any way.

The maximum number of image windows is 50.

1.6 TOOLKIT Commands

1.6.1 How to issue commands

All commands are presented to the user in a menu format; however, any command may be issued from any menu. Commands may be abbreviated as long as the abbreviation is unique. If the command is in the current menu, then the abbreviation only has to be unique within the current menu because the current menu is always searched first. If the command is composed of more than one word, the additional words can be omitted or abbreviated just like the first as long as (a) the abbreviated command remains unique and (b) word abbreviations are separated by a space The case in which the command is entered does not matter.

For example, the command:

```
> Mouse Jagged Line
```

can be abbreviated by:

```
> mouse jagged
> m jag lINE
> m j l
```

etc. but not by:

```
> m      (not unique - will conflict with the ubiquitous
          MAIN MENU if nothing else)
```

```
> mjl   (no spaces between word abbreviations)
```

1.6.2 Command Arguments

Arguments may be included on the same line as the command or may be prompted for (see Argument Prompting below). Arguments placed on the same line as the command are separated from the command by a colon; this is necessary because commands may be made up of more than one word. An example command line is:

```
> Statistics: window_name
```

There are four argument types:

1) text	- printing characters enclosed in quotes e.g. "this is text *(&)(& 1981498"
2) reals	- real numbers identified by a decimal point e.g. -844.4343, 3., .65
3) integers	- e.g. -43 (decimal), 0xff (hex), 067 (octal)
4) strings	- an alphanumeric word beginning with a letter e.g. window1

1.6.3 Argument Prompting

An alternative to specifying arguments on the command line is invoke argument prompting by specifying less than the required number of arguments. For example, if a command required five arguments you could specify the first two and be prompted for the remaining three or specify none and be prompted for all, e.g. entering:

```
> Statistics
```

would result in the prompt:

```
Window?
```

If no arguments are specified a colon is not required.

To quit a command while being prompted for arguments, type a period at the prompt, e.g.:

```
Window? .
```

will get us out of the Statistics command we invoked above.

Argument prompting is also invoked when an argument is found to be an illegal type. (If the argument is the right type but the wrong value, then the command will fail.) All subsequent arguments are assumed to be incorrect as well and are prompted for.

1.6.4 Command Switches

Switches are used to change the action of a command, in contrast to arguments, which instantiate variables employed in the command's computations. Switches are always single letters prefixed by minus signs, except when several switches are concatenated together, e.g.:

```
> Histogram : -t
> Histogram : -t -c
> Histogram : -tc
```

are all legal ways to specify switches.

Once argument prompting takes control, switches cannot be specified!

1.6.5 Multiple Invocations

A command can be invoked several times at once by specifying enough arguments for the extra invocations. For example:

```
> Delete Window : g1 g2
```

will result in Delete Window being executed twice since it normally takes one argument. If we specified:

```
> Make Window : window1 0 0 256 256 g window2
```

Make Window would also be executed twice; argument prompting would take care of the unspecified four remaining arguments.

1.7 Help

The manual that describes the available commands is available on screen, as are brief descriptions of a commands arguments and switches, and brief descriptions of particular menus.

Note that all coordinates asked for and displayed are window coordinates (NOT screen coordinates) unless otherwise specified.

1.7.1 Help on particular menus

If the menu in question is the current menu (ie displayed at the top of the terminal screen), type either ? or ?? at the prompt to get a short one line message describing the menu.

1.7.2 Brief help on a particular command

For a brief list of a commands arguments and switches enter a ? at the TOOLKIT prompt or at an argument prompt, e.g. either:

```
> Histogram : ?
```

or

```
Window Name? ?
```

would display the following lines:

```
Histogram: <source> <t^destination> <c&clip>
Switches: <c = clip hi freq> <t = terminal only>
```

Source, destination, and clip are the arguments, c and t are the switches. The switches modify the provision of arguments in the following way:

o Source is not affected by any switch,
o Destination is only provided if switch t IS NOT specified
  (t xor destination),
o Clip is only provided if switch c IS specified (c and clip).

Thus, if a is an argument and s is a switch:

s^a means s and a are mutually exclusive.

s&a means s and a are inseparable

Entering:

```
> Clear Window : ?
```

Would display the following lines:

```
Clear Window: <window> <b^r [0]> <b^g [0]> <b^b [0]>
Switches: <b = black>
```

A bracketed [] expression following an argument name contains the default value.

1.7.3 Verbose help on particular command

For verbose help that includes a verbal description of a command, its arguments, and its switches, type ?? at either the TOOLKIT prompt or an argument prompt, e.g.

```
> Make Window: ??
Window Name? ??
```

1.8 Scripts

Script files are ascii files containing a series of command lines. These commands should appear in the script file just as if one were typing them in interactively.

A sample script file might look as follows:

```
# this script file builds four windows @ 128x128 pixels
Make Window: $  0    0   128 128 @
Make Window: $  128  0   128 128 @
Make Window: $  256  0   128 128 @
Make Window: $  384  0   128 128 @
```

The $ is used to specify that the user should be prompted for the corresponding argument. The @ is used to specify that the corresponding argument should be the default value. Thus this script will prompt for the names of the windows and assume that the window types are greyscale.

Note also that comment lines are flagged by #.

Scripts can be passed to TOOLKIT as arguments or run interactively. If passed as arguments, specifying the -e switch will result in the script file name and individual command lines being written to the screen. Also, the "more" filter used by TOOLKIT to segment large tty output will not be enabled. If a script is run interactively using the Run Script command, the switch -e will work as above and the switch -i will enable the more filter.

Script files can be created using an editor or they can be created during an interactive session by means of a mechanism that records successfully executed commands. The recording mechanism is turned on by the Script On command and turned off by the Script Off, Quit and Control-C commands.

If a script file contains the Quit command, TOOLKIT will exit.

1.9 Environment

TOOLKIT stores device paths and other such data in environment variables local to TOOLKIT, but displayable and changeable by the user.

Current environment variables and sample values are:

```
Terminal Device   = /dev/tty
Terminal Type     = vt100
Printer           = pr0
Graphics Printer  = gr
Display           = /dev/rs1
Microcode         = /usr/menu/MICROCODE/tk_bmplib.u
Pointer Type      = mouse
Pointer Line      = /dev/tty05
Prompt            = >
Beep On Error     = On
```

1.10 Shell Commands

Shell commands can be entered either at the TOOLKIT prompt or at an argument prompt by prefacing the shell command with the shell escape character !. For example, you could obtain a directory listing before or during the Get Picture command by:

```
> ! ls *.pic
> Get Picture: g0 rabbit.pic
```

or

```
> Get Picture: g0
File Name? !ls *.pic
File Name? rabbit.pic
```

If script recording is on, shell commands issued at the TOOLKIT prompt will be recorded, but not shell commands issed at the argument prompt.

1.11 Summary of Special Commands

cmd	=	command
a	=	argument
s	=	switch
p	=	argument prompt
xyz	=	anything
^	=	control

Symbol	Usage	Importance
:	cmd:a	separates a command from its arguments
??	cmd:?? p:??	displays lengthly command-specific help
?	cmd:? p:?	displays short list of arguments and switches
??	??	displays short help line for current menu
?	?	displays short help line for current menu
$	cmd:$ a	forces prompting for a specific argument
@	cmd:@	tells routine to use a default value for an argument
#	#xyz	precedes comments
!	!cmd p:!cmd	execute cmd in the shell instead of toolkit
^C	^C	causes the terminal screen to be restored to normal, closes script files, exits TOOLKIT
^Z	^Z	causes the terminal screen to be restored to normal, stops TOOLKIT

1.12 Sample TOOLKIT session

(The TOOLKIT program starts up displaying the MAIN MENU:)

```
                                   MAIN MENU
Help            Graphics Menu          System Menu          Main Menu
Quit            Mouse Menu             Transform Menu       Up Menu
Demo Menu       Peter's Menu           Window Menu

============== TOOLKIT Version 1.02, (C) Copyright M.I.T. 1987 ==============

> tran
```

(The user selects the Transform Menu (by typing tran). Incomplete commands are matched and completed. A session using the Transform Menu follows. It shows the use of detailed help (??), help with arguments (?), and argument completion when arguments are not specified. User input is scrolled below the menu. The session ends with a Quit command.)

```
                         TRANSFORM MENU
Binarize                 Integer Operations              Superimpose
Complement               Interpolative Zoom              Thin
DoG Filter               Median Filter                   Threshold
Enhance Contrast         Replace Color                   Translate Pixels
Gaussian Filter          Replicative Zoom                Zero Crossings
Grey Operations          Scale Pixels                    Main Menu
Histogram                Statistics                      Up Menu
============== TOOLKIT Version 1.02, (C) Copyright M.I.T. 1987 ==============

> Repli zo : ??

Replicative Zoom:  <source> <destination> <x zoom> <y zoom>

REPLICATIVE ZOOM: - Does a replicative integer zoom.  The source window is
                    expanded into the destination window by replicating pixels
                    in the X and Y directions.  If the destination window is
                    too big, the extra area is untouched.
Source            - May be greyscale or color.
Destination       - Must be the same type of window as the source.
X Zoom            - Integer scale factor (> 0) for horizontal stretch.
Y Zoom            - Integer scale factor (> 0) for vertical stretch.

> rep zoom : g0 g1 2 2

> threshold : ?
Threshold: <window> <lower bound> <upper bound > <color or grey value>
 Switches: < b = function b> <c = function c> <s = skip color>
> thresh : g2 40 100

> scale
Source? g1
Destination? g2
Scale Factor (real)?  .4

> zero cr : ??
Zero Crossings: <source> <destination>
> zer cros : g2 g3

> quit
```

2. Programmer's Guide To TOOLKIT

2.1 Introduction

This section describes where files are located, how to make TOOLKIT, what the data structures are, how to add and modify commands and menus, and where to find useful subroutines (e.g. for error checking).

2.2 File Locations

/usr/etc	C shell script file "toolkit" used to invoke the correct device-dependent executable file.
/usr/menu/DEMO	Script files for demo-ing TOOLKIT
/usr/menu/DOC	Text of documentation and shell script to create documentation of commands
/usr/menu/INCLUDE	Include files
/usr/menu/MICROCODE	BMP microcode versions of commands
/usr/menu/OLDHIPS	old version hips files used in image file IO.
/usr/menu/SRC	source and object for the infra-structure -- e.g. command parsing, menu tables, etc also the location of the executable files.
/usr/menu/SUBRS	source and object for the commands and for the library (support) routines
/usr/peter/RESEARCH	source and object for some fractal related commands.

2.3 How to Make TOOLKIT

If you have changed an include file, you will have to recompile everything, otherwise you can just go to the directory in which you made the change and type:

```
make all
```

and then you can make the new executables by:

```
cd /usr/menu/SRC
make tk_rt tk_ik
```

The only directories in which you might have to recompile code are:

```
/usr/menu/MICROCODE
/usr/menu/SRC
/usr/menu/SUBRS
```

Remember that if you change a filename or add a file you will have to modify the proper Makefile and possibly delete the old version of the archive library so that no multiple declarations occur.

2.4 The Environment

The environment is a global structure containing environment variables which are loaded from various data bases or set arbitrarily. The actual structure is:

```
typedef struct t_ENV {
        char    *e_term_dev;      /* device driver for terminal */
        char    *e_term_type;     /* terminal type, e.g. vt100 */
        char    *e_printer;       /* printer device */
        char    *e_gprinter;      /* graphics printer */
        char    *e_display;       /* display driver */
        char    *e_microcode;     /* directory containing microcode */
        char    *e_ptr_type;      /* type of pointer */
        char    *e_ptr_io;        /* io line for pointer, eg tty05 */
        char    *e_prompt;        /* command prompt */
        bool    e_beep;           /* beep on error? */
} ENV;
```

The name of global environment is "environment". The global array "env_names" contains printable strings describing each variable. If the structure is modified, then routines in tk_env.c, show_env.c, and set_env.c probably will have to be changed also.

2.5 Graphics Windows

Graphics windows exist on the display device or in core if there is no display device. The structure describing a window is:

```
typedef struct t_WIN {
        int     w_xs;             /* x start - pos of top left corner */
        int     w_ys;             /* y start - pos of top left corner */
        int     w_dx;             /* x width in pixels */
        int     w_dy;             /* y width in pixels */
        char    *w_name;          /* ptr to symbolic name */
        int     w_bpp;            /* bytes per pixel */
        int     w_type;           /* window type */
        int     w_uflags[NUF];    /* array of user flags */
        char    *w_title;         /* title string */
        int     w_us;             /* value of update screen? predicate */
        int     *w_core;          /* ptr to core space for image -
                                     assigned most commonly used type */
} WIN, *WPTR;
```

2.6 Terminal Screen Windows

A terminal screen window is defined by the structure:

```
typedef struct t_SCWINDOW {
        int     _maxy, _maxx;     /* window dimensions */
        int     _begy, _begx;     /* absolute pos of top left corner */
        bool    _scroll;          /* value of can it scroll? predicate */
        int     _outcol;          /* col at which to start writing */
} SCWINDOW;
```

2.7 Menus

2.7.1 Menu Descriptors

The structure describing a menu to be displayed on a terminal screen is:

```
typedef struct t_MENU {
        char    *title;          /* Menu title */
        char    *helpline;       /* Help line for menu */
        int     startx,starty;   /* Starting x and y */
        int     entrycount;      /* Number of entries in menu */
        int     nrows,ncols;     /* Number of rows and columns */
        int     colsize;         /* Maximum size of a column */
        ENTRY   *entry;          /* entries for this menu */
} MENU, *MENU_POINTER;
```

2.7.2 Menu Entries

```
typedef struct t_ENTRY {
        char    *printname;      /* String to print for entry */
        int     submenu;         /* Index of submenu.  -1 if function */
        int     (*f_binding)(); /* Function binding */
} ENTRY, *ENTRY_POINTER;
```

2.8 Commands

Commands work as follows:

1) A command line is entered interactively or by script file.

2) The command line is parsed into the command, switches and arguments.

3) The command is matched with entries in the menu tables until a unique match is found.

4) The menu entry contains a ptr to a function that when invoked returns a pointer to the command data table.

5) The switches are looked up in the commands switch table to see a) if they are legal and b) if they change the number of arguments the command expects (e.g. sometimes the user can specify a switch instead of a particular set of three color arguments).

6) the number and type of arguments are compared with the expected number and type as extracted from the argument table.

7) automatic prompting is invoked if necessary to get additional unspecified arguments or to replace arguments that have an illegal type.

8) the command function pointed to by the data table is invoked or if help is requested the help table and switch info are printed.

2.8.1 Switches

Command switches are contained in an array of switch descriptors. Each descriptor is a structure of the form:

```
typedef struct t_SDES {
        char    *s_name;         /* name of switch */
        char    s_val;           /* value of switch */
        int     s_adiff;         /* change to num of expected args */
        int     s_aindx;         /* NOT USED */
} SDES, *SPTR;
```

2.8.2 Arguments

Command arguments are contained in an array of argument descriptors. Each descriptor is a structure of the form:

```
typedef struct t_ADES {
        char    *a_name;        /* argument prompt */
        int     a_type;         /* argument type */
        char    *a_def;         /* str containing default value */
} ADES, *APTR;
```

2.8.3 Data Table

```
/* structure to hold information about a ms command
*/
typedef struct t_DATA {
        int     (*d_subr)();    /* ptr to func. that performs command */
        int     d_argc;         /* number of args */
        ADES    *d_ades;        /* array of argument descriptors */
        int     d_switchc;      /* number of switches */
        SDES    *d_sdes;        /* array of switch descriptors */
        char    *d_use;         /* short help line */
        int     d_lines;        /* number of lines in long help */
        char    **d_help;       /* long help - array of str */
} DATA, *DPTR;
```

2.8.4 Argument Values

```
/* union to hold an argument in its correct data format
*/
typedef union t_AVAL {
        int     v_int;          /* integer value */
        double  v_double;       /* real value */
        char    *v_str;         /* str value */
} AVAL, *AVPTR;
```

2.8.5 How Data Tables Are Instantiated

In each file containing a command, the programmer explicitly defines an array of argument descriptors, an array of switch descriptors, a single string that constitutes the brief help line, and an array of strings that constitutes the long help. Finally, the programmer calls the macro QUERY with the name of the data table and the name of the subroutine that he wants to add. QUERY expands into the following routine which defines and instantiates the data table. The table is local to the file, as are all the other data structures. If the programmer defines EXPECTARG val, then val is the number of arguments the command will expect to receive if no switches are given. This allows the programmer to specify more argument descriptors in the table than are usually used. Thus a switch s can be defined which increases the number of expected arguments whenever it is specified. Defining EXPECTARG is not necessary to create switches that decrease the number of expected arguments.

```
#define SWITCHTAB       static SDES cmdsubr_s[]
#define ARGTAB          static ADES cmdsubr_a[]
#define USAGE           static char *cmdsubr_u
#define HELP            static char *cmdsubr_h[]
#define SIZES           (sizeof(cmdsubr_s)/sizeof(SDES))
#define SIZEA           (sizeof(cmdsubr_a)/sizeof(ADES))
#define SIZEH           (sizeof(cmdsubr_h)/sizeof(char *))

/* macro defining a function that provides access to a routine's data table
```

```
*/
#ifdef EXPECTARG
#       define QUERY(x,y)\
        DPTR x()\
                {\
                static DATA cmdsubr_d;\
                extern int y();\
                cmdsubr_d.d_subr = y;\
                cmdsubr_d.d_argc = EXPECTARG;\
                cmdsubr_d.d_ades = cmdsubr_a;\
                cmdsubr_d.d_use = cmdsubr_u;\
                cmdsubr_d.d_lines = SIZEH;\
                cmdsubr_d.d_help = cmdsubr_h;\
                cmdsubr_d.d_sdes = cmdsubr_s;\
                cmdsubr_d.d_switchc = (cmdsubr_s[0].s_val ? SIZES : 0);\
                return(&cmdsubr_d);\
                }
#else
#       define QUERY(x,y)\
        DPTR x()\
                {\
                static DATA cmdsubr_d;\
                extern int y();\
                cmdsubr_d.d_subr = y;\
                cmdsubr_d.d_argc = SIZEA;\
                cmdsubr_d.d_ades = cmdsubr_a;\
                cmdsubr_d.d_use = cmdsubr_u;\
                cmdsubr_d.d_lines = SIZEH;\
                cmdsubr_d.d_help = cmdsubr_h;\
                cmdsubr_d.d_sdes = cmdsubr_s;\
                cmdsubr_d.d_switchc = (cmdsubr_s[0].s_val ? SIZES : 0);\
                return(&cmdsubr_d);\
                }
#endif
```

2.9 Adding a Menu to TOOLKIT

Files to Change: /usr/menu/SRC/tk_menus.c
Files to Add: none

1) create a unique #define flag for the menu.
2) create a menu descriptor.
3) create an entry list.
4) add the menu name to the menu list.
5) add a command to the parent menu to access the.
   new menu (this command will require the #define flag).
6) do not forget to include the menu commands:
   MAIN MENU
   UP MENU
   in the new entry list.

If menu descriptors, entry lists, and menu lists are unfamiliar, look at other menus and entry lists in tk_menus.c for an example, and consult section 4 for a description of the structures used.

2.10 Adding a Command to TOOLKIT

Files to Change: /usr/menu/SRC/tk_menus.c
                 /usr/menu/SUBRS/Makefile
Files to Add: /usr/menu/SUBRS/<command>.c

2.10.1 Adding a Command to a Menu

1) Add external declaration of function pointer placed in the entry.
2) Add entry to the appropriate entry list.

For example, to add the command Histogram to the Transform Menu we add the line:

```
        extern int histogram();
```

and the entry {"Histogram", FUN, histogram} to the entry list:

```
        static ENTRY transform_entries[] =
                {
                "Binarize", FUN, binarize,
                "Enhance Contrast", FUN, enhance,
                "Gaussian Filter", FUN, gaussian,
                "Histogram", FUN, histogram,
                "Interpolative Zoom", FUN, intzoom,
                "Main Menu", MAIN_MENU, goto_menu,
                "Up Menu", MAIN_MENU, goto_previous,
                };
```

2.10.2 Creating <command>.c

Use an existing command file (eg histogram.c) as a template. If the command will require a mouse, use m_bread.c as a template.

2.10.3 Modifying a Command To Run on the BMP Processor

```
Files to Change: /usr/menu/SUBRS/<command>.c
                 /usr/menu/SUBRS/tk_bmp.c
                 /usr/menu/SUBRS/xxx_bmp.c
                 /usr/menu/SUBRS/xxx_host.c
                 /usr/menu/MICROCODE/main.g
                 /usr/menu/MICROCODE/Makefile
Files to Add:    /usr/menu/MICROCODE/<command>.g
```

If the command is to be run on the BMP or the host, then use copy.c and copy.g as templates. The layer of indirection exemplified by copy_xxx (see copy.c) serves to isolate code requiring bmp routines, so that bmp code is contained only in the version that is compiled to run on the BMP.

2.11 Useful Support Routines

The following files contain various support routines. All files are in /usr/menu/SUBRS.

core_lib.c	read and write pixels to core
cs_rdhh.c	read image file header
cs_wthh.c	write image file header
drawbox.c	draw a box
drawcircle.c	draw a circle
drawline.c	draw a line
fft_tj.c	compute a fast fourier transform
gsubs.c	gaussian and dog convolution
interact.c	interactively create median filter kernel
mouse.c	mouse io
pyramid.c	move and down multiresolution pyramid
readfile.c	read file into window

solve_lib.c	solve linear matrix equation of form AX=B
spline.c	compute spline
tk_bmp.c	c interface to bmp microcode primitives
tk_rt.c	c interface to Rastech primitives
util_lib.c	transfer image btwn core and display, image transforms
win_lib.c	window manipulation, error checking, assorted other
win_prim.c	read & write pixels - top layer
writefile.c	write image to file
xxx_bmp.c	dispatch to bmp or host depending on hardware
xxx_host.c	dispatch to host only

Add Title: <window> <"title">

ADD TITLE	- Adds a title string to a window's internal representation. Use INFO WINDOW to display title on terminal screen. Use SHOW TITLE to display title in window. Call this routine again to change the title.
Window	- No restriction.
Title	- Limited to the number of characters that will fit in the window. Title must be contained within quotes. One character is 8 pixels wide and 10 pixels high.

Appodize: <source> <destination>

APPODIZE	- filter image with ramp edge in the spatial domain ramp edge defined along outermost ten pixels of shortest radius
Source	- must be greyscale with square dimensions equal to power of 2
Destination	- must be same type and size as source

Bandpass Filter: <source> <destination> <low> <high>
Switches: <i = integer-valued>

BANDPASS FILTER	- delete frequencies below a low threshold and above a high threshold. This filter can serve as a lopass and hipass filter as well.
Source	- must be greyscale with dimensions = power of 2
Destination	- must be same size and type as source
Low Threshold	- lowest passable frequency
High Threshold	- highest passable frequency

Binarize: <source> <destination> <threshold>

BINARIZE	- sets pixels with intensity >= threshold to 255 and sets pixels with intensity < threshold to 0.
Source	- must be greyscale
Destination	- must be same type and size as source
Threshold	- intensity value t s.t. 0 <= t <= 255

Block Fill: <window> <x1> <y1> <x2> <y2> <i^r [0]> <i^g [0]> <i^b [0]>
Switches: <i = inverted color>

BLOCK FILL	- Fills a block with a color. Blocks extending past window boundaries are clipped.
Window	- may be greyscale or color.
X1	- x coordinate of corner 1
Y1	- y coordinate of corner 1
X2	- x coordinate of corner 2 (opposite corner 1)
Y2	- y coordinate of corner 2 (opposite corner 1)
Red	- red color value
Green	- green color value (ignored if window is greyscale)
Blue	- blue color value (ignored if window is greyscale)
-i	- draw block in inverted color Do not provide color arguments with this switch.

Change Directory: <directory>

CHANGE DIRECTORY	- change the current working directory.
Directory	- a UNIX directory pathname.

Circle: <window> <xcenter> <ycenter> <radius> <i^r [0]> <i^g [0]> <i^b [0]>
Switches: <i = inverted color>

CIRCLE	- Draws a circle in a window. Circles extending past window boundaries are clipped.
Window	- may be greyscale or color
X Center	- x coordinate of center
Y Center	- y coordinate of center
Radius	- radius in pixels
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
-i	- draw the circle in inverted color. Do not provide color arguments with this switch.

Clear Window: <window> <b^r [0]> <b^g [0]> <b^b [0]>
Switches: <b = black>

CLEAR WINDOW	- sets every pixel in a window to a color you specify.
Window	- must be greyscale or color
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
-b	- set window to black (0,0,0) Do not provide color arguments with this switch.

Color Threshold: <src> <dest> <"bounds" [see help]> <"colors" [bmvcgyorw]>

COLOR THRESHOLD	- partitions an image into 9 colors. It is used in conjunction with PRINT PICTURE.
Source	- can be greyscale or color but in either case only the red byte is used.
Destination	- must be color and must match the size of the source.
Boundaries	- a quoted list of two to ten monotonically increasing integers that partitions a greyscale intensity range The first integer must be zero; the last must be 256. The default is: "0 30 60 90 120 150 180 210 240 256"
Colors	- a quoted list of characters indicating the color to which the corresponding intensity partition will be set. If n bounds are specified, n-1 colors must be specified. Permitted colors are: (b)lack, (m)ud, (v)iolet, (c)yan, (g)reen, (y)ellow, (o)range, (r)ed, (w)hite The default is: "bmvcgyorw"

Complement: <source> <destination>

COMPLEMENT	- transforms the intensity of each pixel in an image to the difference between the pixel's intensity and the maximum intensity.
Source	- must be greyscale or color
Destination	- must have same type and dimensions as source

Convert: <window> <newtype>

CONVERT	- changes a window's type to the type you specify. Any requisite thresholding must be completed before calling CONVERT, eg grey to binary. Real images are normed to (0,255) before being converted to grey.
Window	- no restriction
New Type	- (b)inary, (g)reyscale, (c)olor, or (r)eal

Copy Window: <source> <destination>

COPY WINDOW	- copies the contents of the source window to the destination window. If the destination is smaller than the source, the source will be trucated. If the destination is larger, then any unneeded area will be left untouched.
Source	- no restriction
Destination	- must be same type as source

Delete Tsave: <window>

DELETE TSAVE	- deletes a tsave.
Window	- may be greyscale or color must be on screen

Delete Window: <window>

DELETE WINDOW	- deallocates a window.
Window	- no restriction

Demo Lambert: <window> <sx> <sy> <sz>

DEMO LAMBERT	- generate image of cylinder using lambert's law Assume orthogonal projection.
Window	- must be greyscale
Source X	- real-valued x position of source
Source Y	- real-valued y position of source
Source Z	- real-valued z position of source

DoG Filter: <source> <destination> <width> <i^dc [0.0]>
Switches: <i = interactive> <o = off center> <p = positive only>
Switches: <n = negative only> <a = absolute value> <s = signed>

DoG FILTER	- Convolves a difference of Gaussians kernel with source. Approximates a Laplacian of a Gaussian. By default the kernel is on center, off surround. Use switch -o to get an off center, on surround kernel.
Source	- may be greyscale or color window If color only the red byte is used
Destination	- If greyscale window, results are normalized to [-127,127] & then translated to [0,254] so that zerocrossings have value 127 If color window, then zerocrossings have value 0 Only color windows can accomodate negative values
Width	- Distance in pixels between poi of inhibitory gaussian
DC term	- Adjusts the volume under the excitatory gaussian Either specified by argument or set interactively using -i
-i	- Set the DC term interactively.
-o	- Change kernel to off center, on surround.
-p	- Show positive values only
-n	- Show negative values only
-s	- Show signs only
-a	- Take absolute value
Note	- The four transformations specified by the switches are performed before the normalization & translation of greyscale destinations.

Enhance Contrast: <source> <destination>

ENHANCE CONTRAST	- maximizes the intensity differences between pixels.
Source	- must be greyscale or color If color only the red byte is used.
Destination	- must be same type and size as source

FFT Image: <source> <destination>
Switches: <l = display log(amplitude)>

FFT IMAGE	- transforms from spatial domain to frequency domain and displays result as intesity image
Source	- must be greyscale with dimensions = power of 2
Destination	- must have same size and type as source
-l	- display logarithm of amplitude rather than amplitude

FFT Plot: <source> <dest> <start> <end> <a^xlb> <a^xub> <a^ylb> <a^yub>
Switches: <a = auto-scale> <f = fit to log(freq)> <p = fit to log(power)>
Switches: <i = integer-valued> <d = display data> <c = clear destination>

FFT PLOT	- plot frequency vs power and show best linear fit
Source	- must be greyscale
Destination	- must be greyscale
Start	- first data point to plot
End	- last data point to plot
X Lower Bound	- lower bound on x data values required to compare plots across images do not specify with -a
X Upper Bound	- upper bound on x data values required to compare plots across images

```
                           do not specify with -a
Y Lower Bound            - lower bound on y data values
                           required to compare plots across images
                           do not specify with -a
Y Upper Bound            - upper bound on y data values
                           required to compare plots across images
                           do not specify with -a
-a                       - scale automatically
-f                       - plot log(frequency) instead of frequency
-i                       - input values are integer-valued, not grey-scale
-p                       - plot log(power) instead of power
-d                       - display data point being plotted
```

```
Fourier Fractal Seg: <source> <dest>
Switches: <d = display slopes>

FOURIER FRACTAL SEG - the Pentland method of doing fractal segmentation.
                      compute fractal dimension for each 8x8 square by
                      means of a fourier transform
Source              - must be greyscale & each dimension must = power of 2
Destination         - must be same type and size as source
```

```
Fourier OH Curve: <source> <dest> <ospace [5]> <start [2]> <end [0]>
Switches: <a = autoscale radius> <d = display curve pts>

FOURIER OH CURVE    - plots curve of orientation vs hausdorff dimension
Source              - must be greyscale & have square dimensions = power of 2
Destination         - must be color
Orientation Spacing - spacing of samples of orientation dimension in degrees
Start               - first data point to plot
End                 - last data point to plot
Radius Scale        - factor to scale radius by
                      Do not provide this argument with the switch -a.
```

```
Frame: <window>

FRAME           - draws a box in inverted color around a window.
                  The box is inclusive, covering the edges of the window.
                  To make the box go away, simply issue FRAME once more.
Window          - may be greyscale or color
```

```
Gabor Filter:   <source> <dest> <file>
                <kxsize [7]> <kysize [7]> <sdevx [2.0]> <sdevy [2.0]>
                <spatial_paving [2]>
                <#octaves [-1]> <minperiod [2.0]>
                <orient_paving [30.0]> <ofirst [0]> <olast [5]>

GABOR FILTER        - transforms from spatial domain to info hyperspace.
Source              - must be greyscale & each dimension must = power of 2
Destination         - must be same size and type as source
Temp File           - unique file prefix for temporary files written to disk.
Kernel X Size       - odd integer defining size in pixels of kernel width
Kernel Y Size       - odd integer defining size in pixels of kernel height
```

```
X Std Dev             - real-valued standard deviation in pixels per deviation
                        defines the x-axis coverage of the gaussian envelope
Y Std Dev             - real-valued standard deviation in pixels per deviation
                        defines the y-axis coverage of the gaussian envelope
Spatial Paving        - rate at which to sample image, eg 3 = every third pixel
Number of Octaves     - Number of levels in pyramid, -1 means use the maximum
                        possible levels.  Since each level is half the size of
                        the level below, levels are each one octave apart
Minimum Period        - real-valued minimum period sampled in pixels per cycle
                        Its inverse is the frequency sampled at the lowest level
                        i.e. the highest frequest sampled.
Orientation Paving    - Spacing of orientation samples in degrees
                        Must divide 180 without remainder.
First Orientation     - index of first orientation
Last  Orientation     - index of last  orientation


Gabor Fractal Seg: <source> <dest> <ofirst> <olast>
                   <kxsize [7]> <kysize [7]> <sdevx [2.0]> <sdevy [2.0]>
                   <spatial_paving [2]>
                   <#octaves [-1]> <minperiod [2.0]>
                   <orient_paving [30.0]> <ofirst [0]> <olast [5]>

GABOR FRACTAL SEG - use gabor filter to compute fractal dimension
Source            - must be greyscale
Destination       - must be greyscale
Kernel X Size     - odd integer defining size in pixels of kernel width
Kernel Y Size     - odd integer defining size in pixels of kernel height
X Std Dev         - real-valued standard deviation in pixels per deviation
                    defines the x-axis coverage of the gaussian envelope
Y Std Dev         - real-valued standard deviation in pixels per deviation
                    defines the y-axis coverage of the gaussian envelope
Spatial Paving    - rate at which to sample image, eg 3 = every third pixel
Number of Octaves - Number of levels in pyramid, -1 means max possible levels
                      Since each level is half the size of the level below
                      levels are each one octave apart
Minimum Period    - real-valued minimum period sampled in pixels per cycle
                    Its inverse is the frequency sampled at the lowest level
                    ie the highest frequest sampled.
Orientation Paving - Spacing of orientation samples in degrees
                       Must divide 180 without remainder.
First Orientation   - index of first orientation
Last  Orientation   - index of last  orientation


Gabor OH Curve: <source> <dest>

GABOR OH CURVE  - plots curve of orientation vs hausdorff dimension
Source          - must be greyscale
Destination     - must be greyscale


Gaussian Expand: <source> <dest>

GAUSSIAN EXPAND - uses 5x5 gaussian kernel to expand repeatedly by factor
                  of 2 from size of source to size of destination
Source          - must be greyscale and dx,dy must be powers of two
Destination     - must be greyscale and dx,dy must be powers of two
```

Gaussian Filter: <source> <dest> <width> <dc [0.0]>
Switches: <n = new version>

GAUSSIAN FILTER	- lo pass filters an image using a gaussian.
Source	- must be greyscale
Destination	- must be the same size and type as source The results are normalized to [a,b], where a & b are the min & max intensities of the source.
Width	- one theory says this is the distance in pixels between poi
DC term	- Adjusts the volume under the gaussian.
-n	- use new version where width = standard deviation in pixels

Gaussian Reduce: <source> <dest>

GAUSSIAN REDUCE	- uses 5x5 gaussian kernel to reduce repeatedly by factor of 2 from size of source to size of destination.
Source	- must be greyscale and dx,dy must be powers of two
Destination	- must be greyscale and dx,dy must be powers of two

Get 1D: <window>

GET 1D	- get fractal dimension for 1d image using FFT
Window	- must be greyscale, uses only top line

Get Picture: <window> <file>

GET PICTURE	- gets a picture from a hips formatted file and fits it into a window.
Window	- no restriction Color windows requires a triplet of files (see below).
File	- name of file matching window type If window is color, .r, .g, & .b will automatically be suffixed to the file name.

Grey Operations: <source 1> <op> <source 2> <destination>
Switches: <n = normalize>

GREY OPERATIONS	- performs binary operations on greyscale windows of identical characteristics.
Source 1	- must be greyscale
Operation	- One of addition(+), subtraction(-), or(|), exclusive or(^), and(&), multiplication(*), division(/), distance(d).
Source 2	- must be greyscale dimensions must match those of source 1
Destination	- must be greyscale dimensions must match those of source 1
-n	- normalize result of operation instead of clipping only use with addition, subtraction, multiplication, and division

Grid: <window> <xspace> <yspace> <i^r [0]> <i^g [0]> <i^b [0]>
Switches: <i = inverted color>

GRID	- draws a grid over a window
Window	- may be greyscale or color
X Spacing	- horizontal spacing between grid lines
Y Spacing	- vertical spacing between grid lines
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
-i	- draw the grid in inverted color. Do not provide color arguments with this switch.

Halftone: <source> <destination>

HALFTONE	- halftones grey areas of an image, ignoring color pixels
Source	- must be greyscale or color
Destination	- must be same type and size as source

Help: NO ARGUMENTS

HELP	- Displays system help.

Histogram: <source> <t^destination> <c&clip>
Switches: <c = clip hi freq> <t = terminal only>

HISTOGRAM	- draws a histogram of the source in the destination and prints a numerical histogram on the terminal screen. The numerical histogram is both unscaled and unclipped. The graphics histogram always has its intensity range scaled to fit the destination window width, and its frequency range is always scaled to fit the destination window depth.
Source	- must be greyscale
Destination	- must be color and at least 256 pixels wide
Percent to Clip	- integer percentage (See -c below) Provide only if using -c.
-c	- Clip the highest x percent of frequencies, e.g. if a total of 10 different frequencies arise in the histogram, clipping the highest 20 percent means that the graphics output will be scaled to maximize contrast between the 8 lower frequencies. With -c you must specify the percentage as an additional argument in integer form.
-t	- Cancels the graphics display. Do not specify a destination window with this switch.

Info Window: <window>

INFO WINDOW provides information about a given window in the format
<Name, X, Y, DX, DY, Type, Updated, Tsaved?>

<Name>	window's symbolic name;

<X, Y>	coordinates of window's top left hand corner;
<DX, DY>	window's width and depth in pixels;
<Type>	window type: Binary, Grey, Color, or Real
<Updated>	tells whether the window is updated on screen or in core;
<Tsaved?>	tells whether or not a window that is being updated on screen has a copy of itself stored in core.

If the window has a title, that also is displayed.

Init Display: NO ARGUMENTS

INIT DISPLAY	- initializes the graphics display device.

Integer Operations: <source 1> <op> <source 2> <destination>
Switches: <g = make greyscale>

INTEGER OPERATIONS	- performs binary operations on greyscale windows of identical characteristics.
Source 1	- must be color or greyscale
Operation	- One of addition(+), subtraction(-), or(|), exclusive or(^ and(&), multiplication(*), division(/), distance(d).
Source 2	- must be color or greyscale dimensions must match those of source 1
Destination	- must be color or greyscale dimensions must match those of source 1
-g	- convert result of operation to greyscale

Interpolative Zoom: <source> <destination> <xzoom> <yzoom>

INTERPOLZATIVE ZOOM	- expands the source into the destination using bilinear interpolation. If the destination is too small, the source is reduced. If the destination is too big, the extra area is untouched.
Source	- may be greyscale or color
Destination	- may be greyscale or color
X Zoom	- real zoom factor > 1.0
Y Zoom	- real zoom factor > 1.0

Life: <window> <birth min> <birth max> <survival min> <survival max> <numgen>

LIFE	- play the game of life
Window	- must be greyscale, must be on screen, and must contain the initial pattern
Birth Min	- minimum number of neighbors for dead->live transition
Birth Max	- maximum number of neighbors for dead->live transition
Survival Min	- minimum number of neighbors to prevent live->dead transition
Survival Max	- maximum number of neighbors to prevent live->dead transition
Num Generations	- number of generations

Line: <window> <x1> <y1> <x2> <y2> <i^r [0]> <i^g [0]> <i^b [0]>
Switches: <i = inverted color>

```
                    The larger the size the better the generated fractal
                    will match the specified dimension.
Scale1            - Size of 1st process lobe unless -t is used, in which case
                    its the threshold in degrees.
Phase1            - Orientation of 1st process lobe.
Power1            - Eccentricity of 1st process lobe.
                    If -t specified, power1 = # degrees sutended by pie slices
Scale2            - Size of 2nd process lobe.
Phase2            - Orientation of 2st process lobe.
Power1            - Eccentricity of 2st process lobe.


Make Window: <name> <x> <y> <xsize [256]> <ysize [256]> <type [g]>
Switches: <c = make in core>

MAKE WINDOW       - allocates a window on a 512 x 512 pixel plane.
                    The top left corner of the pixel plane is (0,0).
                    There are no constraints on position,
                    i.e. overlapping other windows is ok.
Window Name       - a unique symbolic name
X Position        - x coordinate of top left corner in screen coordinates
Y Position        - y coordinate of top left corner in screen coordinates
X Size            - width in pixels
                    The default is 256.
Y Size            - height in pixels
                    The default is 256.
-c                - Update the window in core.
                    Must be used if the Menu System was called with the
                    core switch (-c).
Type              - (b)inary, (g)reyscale, (c)olor, or (r)eal
                    The default is greyscale.


Median Filter: <source> <destination> <xsize> <ysize> <(ai)^"sampling">
Switches: <i = interactive> <a = all pixels sampled>

MEDIAN FILTER     - applies a median filter good for reducing spot noise
Source            - must be greyscale
Destination       - must be same size and type as source
X Size            - horizontal width of kernel
Y Size            - vertical width of kernel
Sampling Scheme   - a quoted string specifying the pixels that the filter will
                    sample to determine the median of the region it covers.
                    The string must only contain ones and zeroes, where a one
                    indicates that an element will be sampled and a zero
                    indicates that it will not. eg: a 3x3 filter that samples
                    the first and third columns would be given by "101101101"
                    and a 5 x 3 filter sampling the first and third rows would
                    be described by "111110000011111".
                    -a & -i also let you specify the sampling scheme.
-i                - Lets you toggle filter elements on or off using a cursor.
                    Do not provide a sampling scheme argument with this switch.
-a                - Indicates that all pixels will be sampled.
                    Do not provide a sampling scheme argument with this switch.


Mouse Block Fill: <window> <i^r [0]> <i^g [0]> <i^b [0]>
Switches: <i = inverted color>
```

LINE	- draws a line in a window. Lines extending past window boundaries are clipped.
Window	- may be greyscale or color
X1	- x coordinate of first endpoint
Y1	- y coordinate of first endpoint
X2	- x coordinate of second endpoint
Y2	- y coordinate of second endpoint
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
-i	- draw line in inverted color Do not provide color arguments with this switch.

List Windows: NO ARGUMENTS

LIST WINDOWS provides information about each window in the format
<Name, X, Y, DX, DY, Type, Updated, Tsaved?>

<Name>	window's symbolic name;
<X, Y>	coordinates of window's top left hand corner;
<DX, DY>	window's width and depth in pixels;
<Type>	window type: Binary, Grey, Color, or Real
<Updated>	tells whether the window is updated on screen or in core;
<Tsaved?>	tells whether or not a window that is being updated on screen has a copy of itself stored in core.

The command INFO WINDOW provides more data about a specific window.
A count of windows used and unused is also provided.

Make 1D Fractal: <window> <dimension> <sample [8192]> <offset [0]>
Switches: <l = line texture> <r = reproducible>

MAKE 1D FRACTAL	- generate 1D fractal texture by filtering 1D real white noise in the frequency domain The output is a graph of position vs elevation.
Window	- must be greyscale
Dimension	- 1.0 <= fractal dimension < 2.0
Sample Size	- the size of the white noise sample Must be a power of two because of FFT. The larger the size the better the generated fractal
Freq Offset	- index of lowest displayed frequency
-l	- generate line texture instead of graph
-r	- generate reproducible random noise

Make 2D Fractal: <window> <dim> <sample [512]> <s1> <p1> <e1> <s2> <p2> <e2>
Switches: <a = one lobed process> <b = two lobed processes>
Switches: <t = threshold process> <o = original output> <z = zeromin output>
Switches: <s = scaledown output> <n = non-reproducible>

MAKE 2D FRACTAL	- make 2D fractal texture by filtering 2D real white noise in the frequency domain. By default the fractal will be isotropic; use the switches to make anisotropic fractals. The texture is presented as an intensity map.
Window	- must be greyscale
Dimension	- 2.0 <= fractal dimension < 3.0
Sample Size	- the size of the white noise sample Must be a power of two because of FFT.

MOUSE BLOCK FILL	- fill a block within a window with a color Use the mouse to select two diagonally opposed corners.
Window	- must be color or greyscale.
Red	- red color value
Green	- green color value (ignored if window is greyscale)
Blue	- blue color value (ignored if window is greyscale)
-i	- draw block in inverted color Do not provide color arguments with this switch.

Mouse Block Read: <window>

MOUSE BLOCK READ	- lets you define a block within a window and examine the coordinates and intensities of each pixel in the block. Use the mouse to select two diagonally opposed corners.
Window	- must be color or greyscale.

Mouse Cross-section: <source> <destination>

MOUSE CROSS-SECTION	- lets you view plots of greyvalue along cross-sections of an image.
Source	- must be greyscale
Destination	- must be color, at least 256 pixels high, and at least as wide as the largest dimension of the source.

Mouse Dots: <window> <r [0]> <g [0]> <b [0]>

MOUSE DOTS	- lets you draw dots in a window using a mouse
Window	- may be greyscale or color
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)

Mouse Drag New Window: <name> <xsize [256]> <ysize [256]> <type [g]>

MOUSE DRAG NEW WINDOW	- allocates a window on a 512 x 512 pixel plane. Use the mouse to move the window frame around the screen by one corner. You can change the corner that the mouse selects. There are no constraints on position, i.e. overlapping other window is ok. The window cannot be created in core.
Window Name	- a unique symbolic name
X Size	- width in pixels. The default is 256.
Y Size	- height in pixels. The default is 256.
Type	- (g)reyscale or (c)olor. The default is greyscale.

Mouse Jagged Line: <window> <i^r [0]> <i^g [0]> <i^b [0]>
Switches: <i = inverted color>

MOUSE JAGGED LINE - lets you draw a an unlimited number of connected line

	segments in a window using a mouse.
Window	- may be greyscale or color
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
-i	- use inverted color Do not specify colors with this switch.

Mouse Make Window: <name> <type [g]>

MOUSE MAKE WINDOW	- allocates a window on a 1024 x 1024 pixel plane. Use the mouse to select two opposing corners of the new window.
Window Name	- a unique symbolic name
Type	- (b)inary, (g)reyscale, (c)olor, or (r)eal The default is greyscale.
Note	- The top left corner of the pixel plane is (0,0). There are no constraints on position, i.e. overlapping other windows is ok. - The window cannot be created in core. - The window is framed after creation; you may remove the box using FRAME.

Mouse Pixel Read: <window>

MOUSE PIXEL READ	- lets you retrieve the window coordinates and color of the pixel under the cursor.
Window	- must be greyscale or color

Mouse Polygon: <window> <i^r [0]> <i^g [0]> <i^b [0]>
Switches: <i = inverted color>

MOUSE POLYGON	- lets you draw a polygon in a window using a mouse.
Window	- may be greyscale or color
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
-i	- use inverted color Do not specify color values with this switch.

Mouse Spline: <window> <i^r [0]> <i^g [0]> <i^b [0]>
Switches: <i = inverted color>

MOUSE SPLINE	- lets you use a mouse to connect knot points with a spline.
Window	- may be greyscale or color
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
-i	- draw curve in inverted color Do not provide color arguments with this switch.

Mouse Straight Line: <window> <i^r [0]> <i^g [0]> <i^b [0]>
Switches: <i = inverted color>

MOUSE STRAIGHT LINE	- lets you draw an unlimited number of straight lines in a window using a mouse.
Window	- may be greyscale or color
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
-i	- use inverted color Do not specify color values with this switch.

Print Picture: <window> <print file>
Switches: <w = write file only>

PRINT PICTURE	- dumps a window's contents to a print file, which is then spooled to a graphics printer. The print file may be deleted any time after is spooled to the printer. Make sure a color ribbon is in the printer.
Window	- may be color or greyscale If greyscale, the printer output is monochrome, otherwise it is color.
Print File	- the name of the file to which the picture will be dumped.
-w	- dump the picture to the print file but do not spool it.

Put Picture: <window> <file>
Switches: <o = overwrite file> <r = write red byte only>

PUT PICTURE	- writes a image to a file on disk.
Window	- no restriction
File	- name of file If the picture is color and -r is NOT used then the file name provided will be automatically prefixed to .r, .g, and .b.
-o	- Overwrite an existing file.
-r	- Write red byte only. Can be used only with color windows.

Quit: NO ARGUMENTS

QUIT	- Resets the terminal screen, closes the scriptfile if it is open and returns you to the operating system.

Random Binary Image: <window>
Switches: <r = reproducible>

RANDOM BINARY IMAGE	- create a random binary image
Window	- must be greyscale
-r	- generate reproducible pattern

Rectangle: <window> <x1> <y1> <x2> <y2> <i^r [0]> <i^g [0]> <i^b [0]>

Switches: <i = inverted color>

RECTANGLE	- draws a rectangle with a color. Rectangles extending past window boundaries are clipped.
Window	- may be greyscale or color
X 1	- x coordinate of corner 1
Y 1	- y coordinate of corner 1
X 2	- x coordinate of corner opposite to corner 1
Y 2	- y coordinate of corner opposite to corner 1
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
-i	- draw the circle in inverted color. Do not provide color arguments with this switch.

Refresh: NO ARGUMENTS

REFRESH - refreshes the terminal display.

Replace Color: <src> <dest> <r1[0]> <g1[0]> <b1[0]> <r2[0]> <g2[0]> <b2[0]>

REPLACE COLOR	- sets every pixel of color 1 to color 2.
Source	- must be greyscale or color
Destination	- must be same type and size as source
Current Red	- red intensity of color 1 - Specify -1 to match any red value.
Current Green	- green intensity of color 1 (ignored if window is greyscale) - Specify -1 to match any green value.
Current Blue	- blue intensity of color 1 (ignored if window is greyscale) - Specify -1 to match any blue value.
New Red	- red intensity of color 2 Specify -1 to leave a pixel's red byte unchanged.
New Green	- green intensity of color 2 (ignored if window is greyscale) Specify -1 to leave a pixel's green byte unchanged.
New Blue	- blue intensity of color 2 (ignored if window is greyscale) Specify -1 to leave a pixel's blue byte unchanged.

Replicative Zoom: <source> <destination> <xzoom> <yzoom>

REPLICATIVE ZOOM	- does a replicative integer zoom. The source window is expanded into the destination window by replicating pixels in the X and Y directions. If the destination window is too small, the source is reduced. If the destination is too big, the extra area is untouched.
Source	- may be greyscale or color
Destination	- must be same type as source
X Zoom	- integer scale factor > 0
Y Zoom	- integer scale factor > 0

Rotate 90: <source> <dest>

ROTATE 90	- rotate image 90 degrees clockwise
Source	- must be square

Destination	- must be same type and size as source

Run Script: <script file>
Switches: <e = echo> <i = interactive> <a = abort on error>

RUN SCRIPT	- executes a script file during an interactive session.
Script File	- script file to run
-e	- Echo script commands to screen.
-i	- Allow user interaction (eg activate more filter).
-a	- Abort on script file error.

Scale Pixels: <source> <destination> <scale factor>

SCALE PIXELS	- scales the value of each pixel by factor ie (pixel = pixel * factor) and then clips the result to [0,255]
Source	- must be greyscale
Destination	- must have same type and dimensions as source
Scale Factor	- real-valued amount by which to scale each pixel

Script Off: NO ARGUMENTS

SCRIPT OFF	- turns off the recording mechanism and prints the number of lines written to the script file.

Script On: <file>

SCRIPT ON	- turns on a recording mechanism that writes each successfully executed command line to a file on disk. The command line will be saved even if you are prompted for arguments. If a command line contains enough arguments to invoke a routine several times then the script file will contain a line for each invocation, eg: entering "tsave: w1 w2" will result in two lines, "tsave: w1" and "tsave: w2" being written to the script file. Command lines beginning with the comment character '#' will be written to the file. However, comments on a line containing an actual command will be disregarded and therefore not recorded.
File	- file to write to If file exists, it is overwritten.

Set Environment: <"variable"> <"value">

SET ENVIRONMENT	- allows you to change environment variables. use SHOW ENVIRONMENT to get the variable names and sample values
Variable	- quoted string naming the environment variable
Value	- quoted string that will become the variable's new value

Show Environment: NO ARGUMENTS

SHOW ENVIRONMENT - displays the value of every environment variable.

Show Title: <window> <xpos> <ypos> <r [0]> <g [0]> <b [0]>
Switches: <v = vertical>

SHOW TITLE	- writes a window's title to the graphics display. Currently clipping is not performed.
Window	- may be greyscale or color
X Position	- x coordinate of top left corner
Y Position	- y coordinate of top left corner
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
-v	- Write the text vertically.

Snap: <window>

SNAP - This a no-op for the RASTECH.

Statistics: <window>

STATISTICS	- find the min, max, mean, standard deviation, variance, skewness, and kurtosis of a window.
Window	- must be greyscale

Subsampling Reduce: <source> <dest>

SUBSAMPLING REDUCE	- subsamples source by factor of 2 to produce destination
Source	- must be greyscale and dx,dy must be powers of two
Destination	- must be greyscale and dx,dy must be powers of two

Superimpose: <mask> <foreground> <background> <destination>

SUPERIMPOSE	- places a foreground over a background according to a mask. If the value of a mask pixel is 255, a foreground pixel is written to destination. If the mask pixel is 0, a background pixel is written.
Mask	- may be greyscale or color
Foreground	- Must be same size and type as mask
Background	- Must be same size and type as mask
Destination	- Must be same size and type as mask

Text: <window> <xpos> <ypos> <r [0]> <g [0]> <b [0]> <t^"text">
Switches: <v = vertical> <t = title>

TEXT	- writes a string to the screen. Clipping is not performed.
Window	- may be greyscale or color must be on screen
X Position	- x coordinate of top left corner of text block
Y Position	- y coordinate of top left corner of text block
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
Text String	- text to print. must be in quoted.
-v	- Write the text vertically.
-t	- Use the window's title as the text.

Thin: <source> <destination> <signal> <background> <threshold>

THIN	- Given a binary image, delete blobs that are smaller than a threshold. Blob sizes are determined assuming 8-way connectivity.
Source	- Assumed to be a greyscale image comprised of pixels that have the the values signal or background. Any pixels having other values are treated as background pixels for the purpose of counting blob sizes but are displayed with their original values in the destination.
Destination	- must be same size and type as source
Signal	- color of blobs to threshold
Background	- background color
Threshold Size	- blob size threshold in pixels

Threshold: <window> <lower bound> <upper bound> <c&greyval>
Switches: <b = function b> <c = function c> <s = skip color>

THRESHOLD	- applies one of three threshold functions to a greyscale image. All functions require that the greyscale range be partitioned into three regions.
Window	- must be greyscale
Lower Bound	- Partitions the first and second regions.
Upper Bound	- Partitions the second and third regions.
Greyval	- Greyval to which regions 2 will be set if using function c. Provide this argument only if using -c.
-b	- Use function b.
-c	- Use function c.
-s	- Skips over pixels that are not greyscale.
Note	- The functions are described below:

```
Function   Switch      Region 1        Region 2          Region 3
                      0 <= x < LB    LB <= x <= UB    UB < x <= 255
-----------------------------------------------------------------------
   a        none       set to 0        set to 255        set to 0
   b         -b        set to 0        unchanged         set to 255
   c         -c        unchanged      set to greyval     unchanged
```

To Core: <window>
Switches: <o = overwrite a tsave> <t = use tsave as contents>

TO CORE	- switches a window from being updated on screen to being updated in core.

Window	- may be greyscale or color must be on screen
-o	- Overwrite the tsave if it exists.
-t	- Use the tsave (if it exists) as the new window contents.

To Screen: <window> <s^xpos> <s^ypos>
Switches: <s = use same <x,y>>

TO SCREEN	- switches a window from being updated in core to being updated on screen.
Window	- may be greyscale or color must be in core
Xpos	- x position of top left corner in screen coordinates
Ypos	- y position of top left corner in screen coordinates
-s	- Use the same top left corner that is already stored.

Translate Pixels: <source> <destination> <translation factor>

TRANSLATE PIXELS	- computes pixel = pixel + translation factor and then clips the result to [0,255]
Source	- must be greyscale
Destination	- must have same type and dimensions as source
Translation	- integer amount by which to translate

Trestore: <window>

TRESTORE	- replaces a window's screen contents with the window's last tsave. A tsave is a TEMPORARY save used to save a window's contents to core in case future changes to the window need to be undone.
Window	- may be greyscale or color must be on screen

Trim: <window> <xthick> <ythick> <i^r [0]> <i^g [0]> <i^b [0]>
Switches: <i = inverted color>

TRIM	- Draws a border around a window, but on the inside. Useful for hiding edge effects.
Window	- may be greyscale or color
X Thickness	- width in pixels of left and right borders
Y Thickness	- width in pixels of top and bottom borders
Red	- red intensity
Green	- green intensity (ignored if window is greyscale)
Blue	- blue intensity (ignored if window is greyscale)
-i	- draw border in inverted color Do not provide color arguments with this switch.

Tsave: <window>
Switches: <o = overwrite old tsave>

TSAVE	- saves a window's contents to core (volatile memory).

	This is a TEMPORARY save used to save a window's contents in case future changes to the window need to be undone. Use TRESTORE to copy a tsave back onto the screen.
Window	- may be greyscale or color must be on screen
-o	- overwrite any existing tsave of the window

Verify Fractal: <source> <destination>

VERIFY FRACTAL	- use second order statistics to see if image is fractal
Source	- must be greyscale
Destination	- must be 512x512

Wedge: <window>

WEDGE	- Replaces a window's contents with a gradient changing column by column from 0 to 255. If there are more than 256 columns in the window the gradient repeats starting at 0.
Window	- may be greyscale or color

White Noise: <window>

WHITE NOISE	- fill an image with white noise in the range [0,255]
Window	- must be greyscale

Zerocrossings: <source> <destination>

```
ZEROCROSSINGS   - applies the following transformations to isolate
                  zerocrossings found by a DoG Filter.
                  +-,-+   ==>  10
                  ++,--   ==>  00
                  0-,0+   ==>  10
                  -0,+0   ==>  01
                  00      ==>  11
Source          - may be greyscale or color
                  If color zerocrossings are assumed to have value 0.
                  If greyscale zerocrossings are assumed to have value 127.
Destination     - must be greyscale.
```